ZED SHAW'S HARD WAY SERIES

Learn MORE PYTHON the HARD WAY

The Next Step For New Python Programmers

ZED A. SHAW

目錄

笨办法学 Python · 续 中文版

原书：Learn More Python 3 The Hard Way

译者：飞龙

自豪地采用谷歌翻译

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

赞助我

协议

引言

> 原文：Introduction
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

流程，创造力和质量。在阅读本书的同时，将这三个词写入你的脑海中。流程、创造力、质量。这本书可能充满了重要主题的练习，每个程序员都应该知道它们，但是从书中获得的真实知识就是这三个词。我写这本编程书的目的是告诉你，我所知道的是，软件中最重要的三个常量。没有流程，你会迷失在想知道如何开始，并有维持长期项目的进展的问题。没有创造力，你将无法作为程序员解决每天遇到的问题。没有质量，你不知道你所做的任何事情是否良好。

教你这三个概念很简单。我可以简单地写三篇博客文章，并说："你去看看，然后你就知道这三个字是什么意思。" 这不会让你成为一个更好的程序员，绝对不是一个可以在未来 10 到 20 年内自食其力的开发者。仅仅知道大概的流程并不意味着，你可以在现实实践中实际应用它。阅读关于创造力博客文章，并不能帮助你弄清如何写出有创造力的代码。要真正理解这些复杂的主题，你将需要将其内在化，最佳方式是将其应用于简单的项目。

当你通过书中的练习，我会告诉你，你将要处理的三个中的哪一个。这是我相对其他书籍的一个改变，其中我试图暗中让你在无意中学习概念。这次我会明确表达，因为很重要的一点就是，把这个概念牢牢地保存在你的头脑中，这样你就可以在练习过程中实践了。然后，你将评估你对应用这个实践的尝试效果如何，以及下一次可以做些什么来改进。本书的一个重要组成部分是，能够客观地反思自己的能力，以及改进自己的能力。在完成一些其他目标的同时，通过专注于一种技术或实践来做到它。

除了流程，创造力和质量外，你还将学习五个重要主题，我认为现代程序员需要掌握它们。这些可能会在未来发生变化，但是它们在至今位置近十年间是至关重要的，因此除非技术发生了巨大的变化，否则它们仍将适用。即使是一些东西，像第六部分中的 SQL，也仍然是相关的，因为它教你如何构建数据，以便在后面不会在逻辑上崩溃。你的次要教育目标是：

- 入门 - 你可以通过执行快速的 Hack 来了解如何启动项目。
- 数据结构 - 我不教给你每一个单一的数据结构，但是让我们开始更完整地学习它们。
- 算法 - 不能处理的数据结构相当无意义。
- 解析文本 - 计算机科学的基础是解析，知道如何实现它，可以帮助你学习编程语言，当他们变得流行的时候。
- 数据建模 - 我将使用 SQL 来教会你，以逻辑方式为存储数据建模的基础知识。
- Unix工具 - 命令行工具在本书中用作要复制的项目，然后你还可以学习 Unix 命令行高级

工具。

在这本书的每个部分，你将一次性专注于三个实践中的一个或两个，直到最后，在第七部分中，你将构建一个简单的网站来应用它们。大作业并不吸引人。你不会学习如何创建下一个创业公司，但他们是很好的小项目，将帮助你在学习 Django 时应用你所熟悉的知识。

完全是个人的事情

许多其他的书被设计为，在团队背景下教你这三个概念。当这些书向你讲述流程时，全部都是关于，如何在项目中与另一个人合作来维护代码。当他们教授创造力时，全部都是关于，如何和你的团队开会来向客户询问问题。可悲的是，这些“专业”书籍绝大部分都没有教授质量。这一切都很好，但是对于大多数初学者来说，这些团队风格的书籍有两个问题：

- 你没有团队，所以你不能练习它们教给你的东西。面向团队的书籍专为那些初级程序员而设计，它们已经有工作，并需要在刚加入的团队工作。在这种情况发生之前，任何团队导向的书籍对你来说都是无用的。
- 如果你自己的个人过程，创造力和质量是一团糟，学习如何在团队中工作有何意义？尽管“团队精神”的粉丝们说，绝大多数的编程任务都是独立完成的，你对自己的技能的评估通常是独立完成的。如果你在一个团队工作，但是你的代码始终是低质量的，并且你不断向团队成员寻求帮助，那么你的老板会给你较低评价。由于它们始终讨论团队有多厉害，当一个初级程序员无法单独工作时，他们从来不会责怪团队。他们责怪初级程序员。

这本书并不关于，成为一个在大企业混日子的优秀程序员。这本书关于帮助你提高你的个人技能，所以当你得到一份工作时，你可以独自工作。如果你改善你的个人流程，那么你会成为一个更强大的团队贡献者。这也意味着你可以启动和发展自己的想法，这是绝大多数项目开始的地方。

第一部分：预备知识

> 原文：Part I: Initial Knowledge
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你需要学习的第一件事就是一切事情。我知道这是吓人的，但我在介绍中提到，你在整本书中只会练习三个技巧。当你完成其他任务时，每项练习都会强化每项技能。我可能会告诉你“制作 `cat` 命令的副本”，但你真正学习的是如何具有创造力。我可能会告诉你“创建一个链表数据结构”，但是你正在做的是将结构化代码审查流程应用到你的编程实践中。这本书的秘诀在于，你只是将项目和练习用作一种工具，来学习三个重要的实践：流程，创造力和质量。

理想情况下，这三个概念并不奇怪。流程只是你用于创建某些内容的步骤。创造力只是你产生和实现想法的方式。质量只是你确保这些实现不是垃圾的方式。如何将流程应用到你的个人开发技能中？如何分析是否已经构建了高质量的软件？如何把一个想法变成现实？所有这三个都是相互关联的，因为你需要一个流程来帮助你获得创造性，然后确保质量，这也需要创造力，因为任何流程都不会始终有效。这是一个出色的、美丽的循环。

完成这本书的过程就是这样：

- 对于书的一部分，我将为你提供锻炼流程、创造力或质量的目标。通常它每次就是两个概念，也可能只是一个。例如，在第二部分中，你通过在 45 分钟的快速会话中创造一些简单工具，来锻炼创意。你也要分析你的起始流程，因为如果你发现难以开始，你将不会很有创造力。
- 每个练习的开始都将为你提供一个提示或目标，以便你在练习中进行思考。每一个这些提示都要求你专注于你正在开展的工作的一个或多个方面。第二部分中的练习 4 给了你简单实现一些东西的任务，然后在练习 5 中，你需要开始列出阻碍你的内容，并尝试消除它们或使其更加高效。其他练习要求你查看你的物理环境，并解决任何分散精力的事情。每次你会考虑这些提示，然后在练习中尝试专注于那个特定的任务。
- 每次练习的结尾都有研究性学习，为你提供更多的挑战。他们可能与该项目有关，或者他们可能更多关于你正在处理的过程，创造力或质量问题。
- 一些练习是“挑战模式”。这意味着我会给你一个工具的描述来实现，通常基于一个现有的 Unix 工具，然后扔你实现它，但没有任何参照的代码。你可能需要先研究一小段样本代码，但通常在这些挑战中没有 Python 代码。这些答案可以在 Gitub 上的 Git 项目 http://bit.ly/lmpthwsolve 中在线获得。
- 其他练习是一些东西的教育性描述，你需要根据我的代码实现它。这些练习将解释一些的东西，例如算法，然后你将尽可能准确地实现它们，并发现任何错误。通常这些练习

将专注于质量，因为会要求你编写自动测试，跟踪错误率，并在“研究性学习”中找到其他问题的解决方案。
- 最后，你将使用实验日志来做笔记，并跟踪你可以用于改善工作方式的指标。我非常明确地将此视为一本日志，它的意思是你的改进的个人账户，你不应该与任何人分享，特别是你公司的经理。这种信息可以用来将你看做工人并压榨你，所以要小心谨慎保管好。

在阅读本书时，你的目标不仅仅是实现几个 Unix 工具的副本。你的目标是使用这些小型 Unix 工具项目，来专注于你处理大型项目的能力。

如果我讨厌你的愚蠢的个人流程，会怎么样？

完全没事。这本书是帮助你成长和改进的东西，所以如果你还没有准备好分析你的工作方式，那么请稍等一下。你可以简单地用自己的方式和自己的时间完成所有的挑战，然后回来并使用你的流程中的约束尝试实现项目。每个练习都代表它自己，个人发展部分几乎适用于你所做的任何事情。做你能做的事情，当你需要处理你的工作方式的时候再回来。

如果我发现自己很糟糕，会怎么样？

这是一个非常实际的可能性，但是我的方法可以帮助你了解为什么你是糟糕的，为了解决这个问题需要做些什么。然后这只是一个需要处理的事情，直到你开始变得更好。将你的日记看管好，没有人会知道你是多么糟糕。然后，当你完成时，你就知道，你到底在什么位置，以及你需要做些什么。不要再怀疑你是一个骗子，或者你不能胜任这个任务。你会客观地认识你的优点和缺点，所以你可以不必担心你在这个世界上的位置。

但是，你可能不像你想象的那样糟糕。这本书的目标是成为一个个人课程，来改善你对你的技能的客观评价。这意味着你应该关注的，不是你在某种程度上有多好，而是要改善多少。如果你发现自己对某个特定练习的表现感到沮丧，那么你需要将其拆解，找出可以改进的东西。你还需要站在其他人的角度上看待这个练习，并客观评估你的改进。专注于改进可以帮助你客观地（不是积极或消极地）思考，并继续学习。

练习 0：起步

> 原文：Exercise 0: The Setup
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你需要设置和配置一些工具来学习此书。有可能你已经有了很多这些东西，但让我们确认一下。

程序员的编辑器

你需要一个程序员的文本编辑器，而不是 IDE。Vim，Emacs 和 Atom 都是程序员的文本编辑器。他们不是简单的文本编辑器，不仅仅能处理文本，而且为你管理整个项目而设计，并能够一次处理大量的程序文件。它们还具有 IDE 中的常见功能，如运行构建命令，脚本，以及其它，但有一个主要区别。IDE 通常与单一语言相关联，因为它对源进行高级的检测，并为你提供编写代码的快捷方式。然后，你不必记住任何事情，只需大多数任何项目中，`ctrl-space` 你的方式。当你拥有 100 个另外的 10x 开发人员，它们编写更多技术债务，你无法处理的时候，这真是太棒了。当你试图学习时，这是一个糟糕的功能。另一个问题是你必须等待某人，为你编写任何新语言的 IDE，因此如果 Microsoft 或 JetBrains 不喜欢某种语言，那么你将被卡住。

你可以使用 IDE 做的一切事情，你也可以使用一个真正的程序员的文本编辑器来做，因为像 Vim，Emacs 和 Atom 这样的编辑器是可脚本化和可修改的，它们永不过时。如果 Haskell ++ 成为下一个热点，你可以同时处理它和所有过去的项目。如果你依赖 IDE，那么你必须等待别人为你解决这个语言。

如果你刚开始想要一个免费的程序员编辑器，那么你应该获取 Atom 或 VisualStudioCode 。这些编辑器运行在我在本书中使用的每个平台上，都是可校本化的，有很多插件，而且易于使用。如果你愿意，你也可以使用 Vim 或 Emacs。

Python 3.6

这本书需要 Python 3.6。在理论上你可以使用 Python 2.7，因为许多练习是没有代码的挑战。不过，这些视频将在答案中使用 Python 3.6，而答案的官方代码库也使用 Python 3.6 。这意味着如果你将答案转换为 Python 2.7，就会有问题。如果你不了解 Python 3.6，那么你可以阅读《笨办法学 Python 3》，来获得基础知识。

工作终端

如果你已经读完了《笨办法学 Python》，那么你知道了我要求你使用终端。现在我不需要告诉你如何开始，但是以防万一，折翼课的视频展示了几个选项。该视频在 Windows 上非常有用，因为微软的终端支持和 shell 脚本的风格发生了巨大变化，现在他们正在支持范围更广的 Unix 工具。

Pip+Virtualenv 工作配置

在本书中，你将安装大量额外的库和软件。在 Python 世界中，这最容易用 `pip` 和 `virtualenv` 完成。`pip` 工具离线安装软件包，并把他们放到你的计算机上，因此你可以将它们导入到你的 Python 脚本中。`pip` 的问题是，你被迫将其安装在计算机上的正式目录中，这需要 root 或管理员权限。解决方案是 `virtualenv` 工具，它在目录中创建了一种“Python 包的沙箱”，然后允许你运行 `pip` ，在这里安装软件包，而不是整个主机。在视频中，我会向你展示如何安装，以及确保你在所有平台安装了 `pip+virtualenv` ，并使用它。

实验日记

在研究项目时，你将要做笔记和记录指标。你会希望得到一个方格纸的本子，或者可能是带有点而不是线的纸张，以及一袋好的铅笔。你可以使用任何你喜欢的东西，但本书中的一部分过程是在计算机外部跟踪事件，在解决问题时，作为改变你的观点的一种方式。你更有可能使用长于计算机的纸张（尽管这可能会在以后发生变化），因此你可能会觉得纸张更“真实”，而电脑没有意义。先把东西写在纸上，然后将它们翻译成代码，这样可以帮助你渡过这个感知上的难题。最后，在纸上绘画更加容易。

Github 账户

如果你还没有帐户，你将需要访问 `github.com` 并注册一个帐户。我将为你提供所有视频演示和所有项目的免费代码，以便你可以检查你的工作。如果你卡住了你可以看看这本书的项目，并看看我如何解决它。还有一些时候，我会让你修复一个项目，我有意留了一些 bug 作为练习。

`git`

如果你有一个 `github.com` 帐户，那么你还需要使用命令行工具 `git` 。`github.com` 将会提供大量信息，有关如何以及在哪里获取它，但请观看视频，来了解如何为你的平台最佳安装它。

可选：屏幕录制软件

这不是必需的，但是如果你可以获得软件来记录屏幕，并且理想情况下，可以同时记录你的脸部，那么这将有助于你分析你的工作方式。我说这是可选的，因为完成记录你的工作可能太困难了，其中你需要检查并分析一些线索，关于如何改善你的流程。我这样做了一段时间，它帮助了我很多，但它也扼杀了我的创造力。我的建议是，如果你负担得起或找到了屏幕录制软件，当你觉得你根本无法弄清楚你做错了什么，需要看观察你自己的工作，那么你应该使用它。我也认为，在工作时记录你的实际的脸部和身体，有助于检查你是否拥有良好的姿势或其他身体习惯，它们可能会使身体损伤，但同样，工作时记录自己一整天太麻烦了。这也是你不能与其他人一起做的事情。

深入学习

这就是你现在所需的一切。随着书的继续，我会在特定的时候指导你所需的其他事情。为了完成这个练习，你现在应该观看你的平台的视频，然后安装我告诉你的所有东西。如果你已经安装了某些东西，那么视频会有一些东西，可以确认你的工具是否正常运行。观看它来确保你可以遵循本书的其余部分。

练习 1：流程

> 原文：Exercise 1: On Process
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

软件开发世界中有两种类型的流程。首先是团队流程，这是 Scrum，敏捷或者极限编程。这些流程旨在帮助一群人合作开发大型代码库，而不会相互干扰。团队流程指的是，每个人如何协调，代码行为标准，报告和管理监督。通常这些团队流程归结为：

- 制作待办事项的列表
- 执行列表上的事情
- 确认正确完成了它们

许多团队流程的错误就是，他们试图控制对个人更好的个人流程。极限编程（XP）流程在这方面可能是最糟糕的，甚至指出，每个程序员都有另一个程序员观察他们的工作，并在文本编辑器显示一些错误的时候向他们喊叫。我强烈反对不在某些教育背景下，将个人流程元素强加于人的流程。它侮辱了我们的专业精神，创造一个独断专行的环境，不会促进创造力或质量。在教育环境中，规定学生使用特定的个人流程方法是必要的，但在工作环境中不是。例如，只有他们是一个初级或新手程序员，需要学习，我才强制某些人结对编程。团队流程应该是这样，每个人都能够工作，但他们需要在所需的质量水平完成工作。

另一种类型的流程是个人流程，我从画家，作家和音乐家那里得到这个想法。作为一个专注质量的创意人士，我正在开发一个流程，帮助你以一致的方式进行工作。事实上，一个业余画家，音乐家或作家的标志，代表了这些人不解它们的流程。通常声称没有创意的这些人实际上相反；他们只是不知道这件事，因此不断把它弄错。大多数其他创意学科都制定策略，帮助他们从概念中创造完成的作品，而不会半途而废。对于画家来说，这是一种方法，将绘画的问题分解为逻辑步骤，来确保更有可能成功。对音乐家来说，这是一个类似的流程，结合了平衡做法，保持他们所选的音乐风格的结构。对于作家，他们的流程是构建自己的作品的一种方式，使其自然流动并且没有漏洞和逻辑不一致性（大多数电视编剧似乎完全没有做到）。

对于软件，你的个人流程需要是某种东西，能够完成以下任务：

- 确定可行的想法。
- 让你开始了解这些想法，看看他们是否会工作，并迅速改变它们。
- 在许多工作会话上逐渐优化你的想法，来避免问题或使你能够轻易恢复。
- 确保你的想法的实现的质量，以便你以后不会被 bug 阻碍。
- 确保你可以与他人合作（如果你愿意）。

请注意，我如何说，你不必与他人合作。自从开源以来，创建软件的概念就包括了对社区的过高要求。如果你不想与别人分享或合作，那么你侮辱了它们的存在，并且被认为是一个反社会的家伙。问题是很少有创意活动是在小组中开始的，通常在小组中开始的创意活动最终都没有创意。这种创意火花通常是，一个或两个人有了一个想法，然后从无到有实现了它。制造一个最终产品，如书籍，电影和专辑，可能需要一个庞大的团队。许多其他的创作活动可以独立完成，如绘画或大多数视觉艺术。

你永远不会找到一个艺术学校，它要求画家只能在团队中创作一幅画。软件不像绘画和写作一样，不是独立的创作流程，没有任何理由是这样。软件是一个模块化的学科，这意味着你可以自己创造一切，而其他人仍然可以使用它，即使他们永远不会和你谈话，永远不会写程序。你可以是一个十足的混蛋，人们仍然可以使用你的软件。写作和绘画都是一样的。有很多品行不端的作家，画家和音乐家，他们仍然受到数以百万计的人的崇拜。

如果你按照个人流程开始工作，有人试图告诉你需要分享，或者你是一个反社会的混蛋，那么他们就是在骂你。人们有权保持隐私，独自一人做自己的事情。要求你为较大项目做出贡献的人，似乎只有那些启动这些较大项目的人，似乎都在赚钱。在这一点上相信我，我已经为软件世界贡献了巨额资金，我还去参加会议，但人们说我不是一个贡献者，因为我没有为他们的项目编写代码（尽管他们从来没有帮我做过一件事情）。

在这本书中，当我说“流程”的时候，我的意思是个人流程。我很少涵盖直接关于与他人合作的任何事情，因为有一堆书已经涵盖了你应该如何与他人合作。有很少的书籍可以帮助你按照自己的个人流程工作，并为你定义处理什么，以及为什么。对于你想要专注的事情，自我为中心、贪婪、反社会或者暴虐都没有任何错误，所以你可以在你所爱的东西上做得更好。

挑战练习

练习实际上是写下你的想法，以及你似乎有什么问题。在这个阶段你可能不知道你的工作方式，因为你不是很有经验。为了帮助你，我编写了一个问题列表：

- 长时间处理项目时有问题吗？
- 你是否倾向于编写有缺陷的代码，而不知道为什么？
- 你热衷于编程语言，但从来没有实现过任何东西？
- 你不记得 API 吗？我也是啊
- 你是否感到自卑或像会被抓住的骗子？
- 你担心自己是否是一个“真正的程序员”？
- 你不知道如何选取一个想法，将其从零开始变成代码？
- 你有入门的问题吗？
- 你在混乱的环境中工作吗？
- 你是否完成了你的项目的第一个实现，不知道如何更进一步呢？
- 你是否在代码的顶部继续插入代码，直到产生了巨大的混乱呢？

想想这些问题，然后尝试写下你在处理项目中所做的事情。如果你没有工作经验，请写下你认为应该在项目中做什么。

研究性学习

- 再写一些这样的问题，然后回答他们。
- 问其他你可能知道的程序员，他们的流程是什么。你会发现他们可能不知道。

深入学习

要记住的是，人们说他们的流程是什么，和他们实际做什么，可能是截然不同的。我们人类倾向于使用一种比现实更积极，更合乎逻辑的方式记住事情。在这本书中，你将打破这种习惯，并使用外部记录的指标（可能是屏幕录像）来确定你做什么。这不是你应该永远做的事情，但是当你提高编码技能时，这是一个很大的帮助。但是，当你询问一些其他更成功的程序员，它们的流程是什么，要记住，他们没有这样做，很可能他们告诉你的不是他们做的事情。如果你能找到一个更有经验的程序员，愿意在工作时记录他们的屏幕，那么比起询问他们做什么，这可能更有启发意义。我建议去观看其他程序员的屏幕录像，只是看看他们如何处理问题并记录笔记。

练习 2：创造力

> 原文：Exercise 2: On Creativity
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

创造力没有什么特别之处。如果你是一个智力平均或以上的人，那么你拥有创造力。能够让你的想法变成现实，只是人类智慧和思想的一个方面。问题是创意已经成为特殊类别人士的一张电话卡，叫做《The Creative》。有整个一本书，描述了艺术世界的这个神话牧师，他可以想象出一个概念，并用他们疯狂创意的点金术，制作纯粹的情感上的、智慧的、投入感情的艺术作品，使天上的宝宝们的泪水变成纯粹的白金。坦白地说，“创造力”这个词是一个过度使用的陈词滥调，用于将人们从想法的实现中分离，但我别无选择，只能在这本书中使用这个词。

在我的书中，“创造力”一词只意味着“形成一个想法，并在现实世界实现”。我并不是指这个词的任何优越性，对于擅长实现思想的人也不会有任何神奇的意义。我是一个据说非常有创意的人，我和你之间唯一的区别，是我已经实践了我所拥有的想法，使他们变成现实。我拥有一个想法的笔记本，并尝试定期实现它们。我学习绘画，音乐，写作和编程，作为实现想法，以及在现实创造它们的手段。仅仅通过尝试定期创建一些东西，我已经变得擅长这样做，而且没有任何魔法。我只是不断尝试，直到我能做到。

创造我已经在大脑中拥有的东西的学习过程，已经产生了史诗般数量庞大的垃圾堆，但在垃圾堆的顶部是我欣赏的一些作品。如果你想要使用你的创造力，那么你也必须自己制作垃圾堆。但是，你不能随便地创建一堆垃圾，并希望在你到达顶部时变得很棒。成为有生产力的创意人士的技巧是，在一个流程或一系列约束条件下，学着实现你的想法，引导你走上学习的道路，但要避免严格流程的陷阱，它会抹杀你的创造力。一个有想象力的人的平衡法是一条线，这条线在引导你的流程和杀死你想法的流程之间。我希望在这本书中你会发现那个最佳位置。

挑战练习

要进行你的创意流程，你首先需要随机处理一些东西。我认为我的主要优点之一是能够采取两个看似随意的想法，并把它们变成有趣或有用的东西。你可以每天做这个小小的练习来开始工作：

- 写下至少三个随机组合的单词。愚蠢的森林鼠蜥。 象征主义的法式薄饼。Python 可以召唤外星人。

- 然后花10分钟写一篇关于这三个词的文章，或者其中一个，通过尽可能多的你可以想象到的感官 - 视觉，声音，平衡感，味道，气味。查看人类实际上有多少不同的感官，来了解你可以写的内容。不要自我审查，只是让这些话流出来。你也可以画出想法，绘画或写诗。
- 在此期间，你可能会突然拥有实际想法，它们与软件或其他感兴趣的主题相关。将它们写在更显眼的位置，以便后续探索，甚至可以绘制它们。

无论相信与否，在你坐下来实现软件的时候，这个简单的小小练习会改善很多事情：

- 它教会你让你的想法流动，而不是审查他们。
- 它训练你自由地将看似不连贯的想法联系起来找到可能的连接。
- 它打开了你自己的想法，没有自我批评。
- 它提高了你在写作或绘画中，表达自己想法的能力，这通常是将想法变成现实的第一步。
- 它迫使你想象你的感觉如何工作，以及他们如何为其他人工作，这有助于你在现实世界中实现它们。
- 它也使人们认为，你是大牛和艺术家。你也可以自己去买一个贝雷帽，并在此之后搬到巴黎。

这个随机写作和思考荒谬概念的过程，对于那些习惯于专注软件细节和担心质量的人来说，可能是困难的。这是完全可以理解的，你肯定还需要你形成的这种质量意识。没有严格的质量意识的创造力只会产生垃圾。然而，没有创造力的质量缺乏必要的想象力，来查看你创造的东西错在哪里。你需要的是创造力和质量的混合，可以帮助你创建软件并确保其健壮。

深入学习

如果写下随机单词的想法是这样的，像是“Unitarians tend to fly omelets”，那么你可以从字典中简单地选择一个随机单词，并使用你的感官来写出来。这样做也是一样，也不会很无聊，但我会鼓励你随意一些。没有人因为在珍珠海岸上创作有关金色蜜蜂的诗歌而被解雇。另一个选择是从所有感官的角度，来表达你的感受。这也可以使你有创造力，而且有益于健康。

练习 3：质量

> 原文：Exercise 3: On Quality
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

我将提出一个关于认知的科学理论，我并不能证明它：

> 你所做事情的记忆，会让你思考最终产品，这是正确的行为。

这基于我所做的，几乎每一个创造性的事情的观察，它是这样：

- 你创造的东西需要很长一段时间。这可能是软件，绘画，写作或任何需要时间的东西。
- 你“完成”了它，然后当一个朋友过来之后，退后一步并且惊叹于它有多好。
- 你的朋友然后指出了一个明显的问题，突然间，你对你所创造的东西改变了看法。
- 现在你看到的是你朋友指出的这个错误，你不知道怎么可能错过了它。

我相信这种现象会发生，是因为你记得你如何实现了它，它影响着你所认知的概念。创造的行为往往是一个积极的思想和工作流，所以你的回忆更积极或中立。这样就会使你对作品的看法变得模糊，让你认为它比实际更棒，但也隐藏了许多缺陷和细节。自从你创造它，并且记住了所做的事情，还有一种对作品的情感依恋，这影响了您对作品的判断。然而，你的朋友没有任何回忆，能够更客观地看待作品，这使其更容易看到这些缺陷。这就是为什么复制编辑者发现的错误比作者多。或者，为什么安全专业人员比作者发现更多的缺陷。

在绘画世界中，这是很常见的，画家有很多技巧来颠覆现象。莱昂纳多·达·芬奇（Leonardo Da Vinci）在他的笔记本中甚至提到了这些技巧，他们的设计目的是，让画家站在他们挑剔的朋友的角度上：

- 将画面颠倒下来，从更远的地方看。这表明了颜色和对比度的明显问题，同时也显示出您需要改变的重复形状。在一个优秀的作品中，重复的形状是不受欢迎的。
- 在镜子中看着一幅画，将其水平翻转，所以你的大脑没有如何创造的概念。水平翻转将它变成一个你从未见过的全新的画，然后突然间你是一个讨厌的挑剔的朋友。
- 通过红色玻璃或黑色镜子来看这幅画，它可以去除颜色，使其只能以黑色和白色显示。这显示了绘画太亮或太暗的区域，这使得它在颜色上看起来奇怪。
- 通过放在他们前额上的镜子来看待绘画和主体，向上看镜子，使镜子和主体上下翻转，以便比较两者。这显示了绘画的明显问题，并使场景和绘画看起来像抽象的形状，你的大脑没有记忆。
- 把画放几个月，所以你忘了你怎么做了，然后再次看它。
- 请你讨厌的朋友看看它，让他们告诉你他们看到什么。

一些画家甚至在他们的画后面放了一面镜子，所以他们可以简单地转过来检查他们的进度。我经常使用黑色镜子（或者只是将你的手机的屏幕关闭）放在我的额头上来检查绘画。

在其他创意学科中，没有这么多的自我批评技巧，并且在软件中也只有很少。事实上，我发现程序员由于他们使用“程序员的方式”来完成代码而声名狼藉。“程序员的方式”，指程序员 Hack 一小段代码，然后改一改，直到通过编译，之后宣城他们完成了工作，并继续。事实上，在这之后有很多事情要做，例如清理代码，执行质量保证检查，添加不变量和断言，编写测试，编写文档，并在整个系统的大环境中确认是否工作。但是没有，程序员经常在编译器（或测试套件）没有错误时就停止了。

在这本书中，你将学习如何执行自己的一套检查，类似于画家使用的检查。他们是看待你的代码的方法，并断开了你如何制作它的历史，秘密就变成了检查清单。颠覆你的工作记忆的方式，是强制自己遵循一套检查，它假设你写的东西有缺陷。我交给你的质量过程不会捕获到所有的东西，但它会帮助你发现你能发现的，尽可能多的错误，也可以帮助你跟踪正在犯下什么样的错误，所以你可以在将来避免他们。之后，我会鼓励让其他人审核您的代码，并审核其他人的代码，以便您可以擦亮眼睛，找到更多的缺陷。

缺陷减少的理念是一种概率。你永远不能删除所有的缺陷。相反，您将致力于降低出现缺陷的可能性，并能够粗略估计其概率。这样可以避免您不知道您的代码是否有缺陷，并帮助您摆脱，不知道你的代码是否存在缺陷的恐慌。你不再使用“程序员的方式”，反之，当你完成并准备复查时，你会拥有很好的概念。你不再不断地担心每一个不可能的边界情况，你将能够评估这些边界情况的可能性，并处理最可能的情况。

研究性学习

在这个练习中，你需要找到一段你在几个月前写的代码，然后再回顾一下。您可能不知道如何审核一段代码，但只要浏览代码，并对任何您不喜欢的内容编写评论即可。关键是逐行并逐个文件查看每一行代码。然后，标记您发现的令人反感的代码，并写出原因。它不一定是一个非常大的软件，只是你之前写的一些东西。

深入学习

编写您发现的所有缺陷的列表，并尝试对其进行分类。您可以查看正式的缺陷类别，但是一个很好的基本集合是：逻辑，数据类型，调用。逻辑错误是，你写的 `if` 语句或循环是错误的。数据类型错误是，您使用变量，并假定它是错误的类型。调用错误是，你调用一个函数并且犯了错误。这些不是正式类别，但是这是一个很好的开始。

第二部分：简单的黑魔法

> 原文：Part II: Quick Hacks
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你有最好的想法，你会打动世界！你会成为一个亿万富豪！你的大脑与这个概念融为一体，你会在你的梦想中看到它，它像幽灵一样困扰着你。下一步就是实现它，将其从大脑中移出并放进计算机。你必须杀死鬼魂，将幽灵从灵魂世界中带出来，将其绑定到一个 Python 图腾上，并将其扔到互联网的海洋中。

对你而言足够有创意嘛？

创造力的敌人就是起步。如果在安装程序的过程中存在障碍，你怎么能实现你的梦想？如果你的想法非常强烈和庞大，让你开始担心呢？你足够好吗？你足够聪明吗？那个著名的程序员，会首先让你写出测试，否则会由于你不知道该怎么做而生气嘛？入门通常是创造力中最困难的事情之一，这本书的这部分旨在让你渡过它。

我是画家，音乐家，作家和程序员，所以我知道一两件关于创意的事情。我更了解入门和流程。当我对处理某个项目不感兴趣的时候，流程是将我拖出项目的泥泞的东西。但是，如果没有首先的起步，我就不能走到这一步。

起步需要勇气，并且对任何人的想法有些不关心。在绘画中，当我不能开始时，我只是随意抓取颜料，并在画布上砸在正确的位置。很多非常成功的画家都是这样工作的。其他画家从头开始研究 - 学习，测试，素描，然后最终将它们全部放到一起来开始。作为一个作家，我所做的第一件事就是在我的房子周围散步，疯狂地和自己说话，想像我正在和某人交谈，然后当我说出足够的说话时，我坐下来写作。我只是写下了出现在我脑子里的第一件事。

我不会坐下来写作，或者担心语法。我不会问，“我很聪明吗？”我只是写出我如何说话，并将其输入键盘，然后当我完成了几个段落，我会看一看。它是否有意义？我需要清理吗？这样的确有效，并让我继续下去。也许我写的是垃圾。但是，我开始了，并且这个很重要。之后，我依靠我的流程把起始点变成一个完整的词。

你如何看待这个创造性的起步？我的朋友，你需要弄清楚，并且这本书会有所帮助。首先我们需要打破你对这个起步的恐惧。也许甚至不再害怕了。也许在你开始编程之前，你需要做大量无意义的事情，这样你才能够溢出所有障碍。

我在第一部分说过，如果你每次去做一件事的时候，你必须清理一小时，并修复一堆工具，你就不会有创造力。

如何实践创造力

在这本书的这一部分，你将通过让自己立即开始，快速实践创造力。我会给你简单的超级无聊的小项目。我的意思是，Unix 的 `cat` 命令只是输出一个文件。老实说就像两行 Python 一样简单。这些项目的起步是重要的，为了起步，你会变得冷酷无情。你要坐在你的电脑旁边，纵身已阅，让事情发生。就是现在，不是30分钟后。

你怎么做到他呢？你需要一个清单，你需要自动化。清单是你必须准备好去做的所有事情。打开你的电脑，关掉社交媒体，启动你的编辑器，摸一摸你的幸运的橡皮小鸭，做一个神圣的祷告，冥想 10 分钟，然后复制你的项目框架并开始。这是一个例子，但你需要一个检查列表，并且越短越好。

但你不知道这个清单是什么。也许你有一个想法，但你在开始工作之前真的知道你所做的一切吗？这就是每个项目中，你将重点关注的东西。在第一个项目中，你会坐下来尝试一下，但你会记下你所做的一切。你不能管理无法测量的东西，这是衡量自己的第一步，来看看你如何做某件事情。如果你有屏幕录制软件会更好。打开它，并记录你自己写出了一个糟糕的软件，然后观看视频。记下你做了什么

为了确保你不要在项目中埋头骨干，而是练习起步，你还将为每个项目设置一个严格的计时器。你必须在45分钟内完成最棒的垃圾。不多不少。开始时设置一个45分钟的定时器，准备好你的本子和铅笔，然后去做吧。当定时器关闭时，你完成了。看看你做了什么，然后好的那部分才会出现。

每个项目后，拿起你的列表，并找出可以做什么来消除障碍。你坐在那里，做许多小文件，并且必须在互联网上查找嘛？制作项目框架吧。你在文本编辑器中输入命令时似乎有问题吗？花时间学习来更好地使用它，或学习盲打。你不知道基本的命令和 API 吗？去获取一些书来学习吧，我的朋友。

然后删掉代码并重新开始。从头开始。使用新的纸张开始写，或开始录制。无论你需要做什么来跟踪你所做的事情。这次你更进一步了嘛？有更少的障碍吗？你的目标是减少想法和实现之间的时间，直到起步只是你做的意见事情，就像吃饭和呼吸那样，最终你会感觉到起步是自然的，之后你可以继续下一个项目。

请记住，你要立即坐下来并编程。尽管去做吧。如果一个内心的声音告诉你，你这样做错了，就告诉那个愚蠢的声音来让它闭嘴。这才是 Hack。保持放松并全力以赴，就像你只是把代码扔给一个朋友，它知道你是傻瓜但仍然很有趣。像测试和质量这样的愚蠢的事情可能会在本书的后面出现，但现在只需要编程，把东西搞乱，实现一些黑魔法。获得想法比赢得虚构的质量比赛更重要。

在每个仓促的 45 分钟 Hack 之后，你坐下来查看你是怎么做的。这个“创造然后批改”的流程，将来可以帮助你改善。

菜鸟程序员的流程

如果你刚刚开始，并且在启动项目时还是完全失败了，那么我将为你提供一个简化的流程，以便你使用它来开始。这部分的练习是进行 45 分钟的 Hack（编程），但作为一个菜鸟程序员，你可能需要更多的时间，或者你可能不知道从哪里开始。在这种情况下，请随意使用 60 分钟或使用两个 4 5分钟的时间进行每项练习。

对于一个流程，菜鸟程序员应该在每个会话之前执行以下操作（在启动计时器之前）：

- 准备好你的电脑，并确保你准备好了。
- 阅读任务描述并记下笔记。这是你的研究阶段，你需要以书面形式收集尽可能多的信息。
- 进行研究，并将其转化为 TODO 列表，了解你需要做什么来实现这个黑魔法。写下你可以想到的任务。你需要创建什么文件？什么目录？什么功能？你会使用哪些库？

一旦你有了 TODO 列表，你就可以启动定时器了。在你的 Hack 会话中，你将执行以下操作：

- 选择你的 TODO 上第一个最简单的任务，并完成它。你需要一个文件吗？创建它！你需要一个目录吗？创建他！
- 检查你刚刚做的工作。
- 删除当前任务，执行下一个任务。

我认真对待这个流程。它是我使用的一个较小的版本，但它可以用。几乎每个流程都只是“制作一个清单，完成它，检查它”。如果它适用于我，它将适用于你，所以如果你不知道该怎么做，你应该使用它。

菜鸟程序员的编程流程

此过程也将适用于你编写的代码。我在第一本书“笨办法学 Python”中介绍了它。当你不确定如何写一段代码时，遵循这个流程：

- 用简单的中文写出，你的代码应该做什么。如果你需要把它写成一个段落，那么就这样做。如果你可以把它写成一个任务列表，那就更好了。如果你写出了一个段落，那么你将把它转换成代码必须做的事情的列表。
- 把这个列表变成注释，把 `#` 放在每行的前面。
- 从顶部开始，在每个注释下，编写 Python 代码使其有效。如果注释太抽象，那么将其分解成较小的注释并重复此步骤。
- 运行代码，来确保你刚才写的内容没有语法错误，并且基本能工作。

这就是你需要做的所有事情。如果你可以用你的中文（或任何自然语言）来表达你想要的代码，那么你可以轻松地实现代码，而不必考虑代码。最终你不需要先写注释再写代码，但是当我卡住的时候我还是这样做的。

练习 4：处理命令行参数

> 原文：Exercise 4: Dealing with Command Line Arguments
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在你能处理本书的第一部分之前，你需要完成一些简单的黑魔法，教你如何使用 Python 中的命令行参数。

传统上我们称这种黑魔法为“spike”。该术语来自于一个小型测试项目，涵盖了更大的流程或项目的所有元素。这个小型测试黑魔法“spike”通过一切手段，来确保你可以使用它。spike 的目的是，通过排练来了解如何使用一些新的库或工具，然后真正在你的项目中使用它。

这也是第一个具有“挑战性”的练习。挑战性旨在让你了解如何做某事，然后你可以看看我如何做的，并与你的工作进行比较。我不会先给你代码，并且你需要把它写出来。你不再是初学者了。你现在正在阅读一个挑战，然后你必须解决它。

> 警告
>
> 仔细阅读这个警告！你不应该在 45 分钟内完成完整的发行版软件。45 分钟的时间限制是让你尽快开始，并且不要担心你会做错事情。它是你工作的助推器，而不是测试。这意味着，如果你遵循 45 分钟的时间框架，然后暂停，因为你认为，你不能完成一个伟大的、美丽的作品，你就是做错了。你应该看看，“我们来看看我能在45分钟内完成什么”。这些练习是开放式的，因为不同的人在给定的时间内完成不同的工作量。你只是使用时间约束，来找出你的工作方式，而不是弄清楚你是一个糟糕的程序员还是一个伟大的程序员。

挑战练习

你要编写两个小的 Python 脚本，它们使用两种方法来测试处理命令行参数：

- 普通的旧式 `sys.argv`，像往常一样。
- Python 的 `argparse` 包，用于更花式的参数处理。

你的测试脚本应该能够处理以下情况：

- 通过 `--help` 或 `-h` 获得帮助。
- 至少有三个参数是标志，这意味着它们不需要一个额外的参数，只是将它们放在命令行上就可以了。

* 至少有三个参数是选项，这意味着，它们会在你的脚本中接受一个参数并将一个变量设为它。
* 额外的“位置”参数，这是文件的列表，在所有 `--` 风格参数的末尾，并能处理终端通配符 `*/.txt` 。

因为这个练习是一个 spike，你应该这样，如果你在测试过程中感到痛苦，你可以放弃它并尝试另一件事。开始尝试用 `sys.argv` 解决这个问题，然后如果你不能想出来，尝试使用 `argparse` 。

记住，这是一个45分钟的定时练习，你需要坚持下去。你还必须跟踪你做的所有事情来开始。这个练习的目的是，弄清楚如何继续以自己的方式开始一个项目。甚至在你开始之前，你是否在劝阻自己？你是否不知道你的文本编辑器在哪里或如何使用它？写下来，然后弄清楚如何去除这个障碍。

但是，不要将这个严格的45分钟练习与失败混淆。你正在尝试在45分钟内做任何事情。如果你的技术水平是这样的，你完成了一个 `ex4.py` 文件，没有别的，那么你在45分钟内做了一些事情。然后，你应该看看为什么所做的一切都是开始编写这个文件，弄清楚下一步需要做什么，然后尝试另一个45分钟的流程。

答案

为了防止你作弊，所有答案的代码都在本书的项目网站 http://bit.ly/lmpthwsolve 上，该网站托管在 https://github.com/ 上。而不是将代码包含在这里，所以你想作弊的时候，只是稍微看一下它，你就必须去查看项目，并访问 `ex4` 目录，看看我是如何实现这个黑魔法的。你还会发现我的笔记，我是如何开始，以及改进。

> 警告
>
> 如果你遇到了障碍，回到第三部分，使用我提供的 Early Coders 中的流程。你需要列出一个清单，执行它，并检查你所做的事情。就是这样。

研究性学习

* 有多少其他的 Python 参数解析库？有没有你喜欢的？
* `argparse` 与 `sys.argv` 相比，主要优势是什么。
* 在项目启动方法中，你可以改进什么？有什么可以去掉的事情吗？

练习 5： `cat`

> 原文：Exercise 5: cat
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

做完练习 4，你开始了解阻碍你的东西。spike 研究是一个简单的挑战，即如何从用户获取命令行参数。练习的真正目的是记录实验笔记，了解你需要做什么来开始。你有没有发现需要改变的东西？任何奇怪的习惯或配置问题？在这个练习中，你将创建一个名为 `cat` 的简单命令的副本，但你的真正目的是选择一件事来更改你的配置，以便你更快入门。记住，关键不是你的 `cat` 的实现。而是你的起步有多快，以及在45分钟内做一些有用的事情。

与以前的练习一样，坚持45分钟的最后期限。为你在练习上花费多少时间设置限制，是让你进入编码模式的有用技巧。事实上，如果你每天花45分钟的时间来热身，这是今后的理想实践。在你可以做到它之前，你需要一个更好的开始，所以找到你今天破除的障碍，让我们开始吧。

> 警告
>
> 我会再说一次，使之更清楚：你不能在这个练习中失败。如果你将45分钟的限制看做一个分级练习，并存在你应该或不应该做的有多好的任何期望，那么你就错了。理解这 45 分钟的最佳方式是，它仅仅是一个机制，在后面踢你，让你往前走。这不是一个考验。我重申，这不是一个考验。不断告诉自己，放松下来，尽管去做吧。

挑战练习

`cat` 命令是“链接”的缩写，它最常用于将文件的内容转储到屏幕上。这样使用：

```
cat somefile.txt
```

该命令输出 `somefile.txt` 的内容。这实际上并不是原始目的。最初是用于组合多个文件 - 因此它被称为 `cat` 。为此，只需将每个文件添加到 `cat` ：

```
cat A.txt B.txt C.txt
```

然后， `cat` 命令遍历每个文件，将其内容输出出来，然后在遍历完所有文件时退出。问题是，这怎么能连接文件？为此，还需要使用终端中找到的 POSIX 文件重定向功能：

```
cat A.txt B.txt C.txt> D.txt
```

你应该熟悉 `>` 符号的用法，如果不是，那么你需要复习基本的 Unix shell 操作。它只需要 `cat` 命令的标准输出（在这种情况下，它是 `A.txt B.txt C.txt` 的全部内容组合），并将其写入右侧的 `D.txt` 文件。

你需要尽快重新实现 `cat` 命令，使用你从练习 4 中学到的命令行参数。请记住，要执行标准输出，只需使用 Python 中的 `print` 即可。要了解更多有关 `cat` 的信息，请使用 `man` 命令：

```
man cat
```

这是 `cat` 命令的手册，在45分钟内，你可以得到尽可能多的实现奖励积分。

答案

你可以在 github 的 http://bit.ly/lmpthwsolve 仓库中找到我的解决方案。它在 `ex5/` 中，你会看到我完成了一个相当简单和肮脏的解决方案。如果你开始这个练习并且担心质量或创造力，那么你这样做是错误的。你的任务是马虎，快速，使之完成。时间限制的关键是，让你摆脱这个想法，每次按下键盘，都必须写出完美的代码来崇拜。尽你所能去做，然后之后你可以分析它并看到改进的地方。

研究性学习

- 有没有任何 `cat` 的惊人功能，你从未使用或难以实现？
- 你能从你的启动流程中破除一个障碍嘛？这比实现 `cat` 更重要，所以如果你没有破除这个障碍，那么你需要再做一遍这个练习。
- 你能找到更多阻碍你的东西吗？简单的事情，如你的颈椎病，因为你座位太低了？没有一个很好的键盘？你的精神状态怎么样？有没有你以为阻止你的东西？你能停止思考它们吗？

深入学习

这不是一本自助手册，我不会修改你的心智，但我发现，学习新事物的巨大障碍，不是主题，而是你的恐惧。如果在这个练习中发现，缺失的想法或恐惧让你无法开始，那么我建议你在实现 45 分钟的黑魔法之前，先花 10 分钟记录你感觉如何。写下你的恐惧，焦虑和感觉将表达他们，并帮助你了解，担心一些事情是非常不实际的，因为它们就像 45 分钟的黑魔法那样简单。尝试一下。你会感到惊讶的是，关于你的感觉的 10分钟的写作会改变你的感觉。

练习 6： `find`

> 原文：Exercise 6: find
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

希望你正在发现阻碍自己的各种方式，甚至在你开始工作之前。也许这不是戏剧性的，但你至少应该确定你在环境中可以改善的东西，它们使你难以开始工作。这些小练习是你专注于开始的一个好方法，因为它们不是那么重要，并且能够放进一个适合你分析的时间尺度。如果这些项目有几个小时那么长，当你复查你做了什么，并做出改进的时候，你会觉得无聊，短暂的45分钟的项目，是你可以记录和非常快速审查的东西。

这是我在学习中使用的一种模式。我会识别我需要改进的东西，例如我如何开始，或者我如何使用一个工具。然后我会设计一个练习来关注它。当我正在学习画画时，我外出并努力画树。我坐下来看着问题，我发现的第一件事就是我拖住了太多的东西。我也把我所有的东西都放在我公寓附近的随机的地方。我为我的绘画用品购买了一个特定的包，并保持这个包是准备好的。当我想在外面画画时，我拿起这个包，走到几个地方之一，而不是精心规划到哪里绘画。我练习拿起我的包，走到两个地方之一，准备好，画一幅画，然后回家，直到我像丝绸一样平滑。之后我看了 Bob Ross 的作品，弄清如何画树，因为这个家伙可以很快画出来很多树。

这是你应该做的。很多人在他们的工作领域浪费时间和精力。你是否有一个专门的工作场所，永远不会改变吗？我放弃了我的笔记本电脑，现在只需要使用台式机，这样我就可以在一致的地方做我的工作。这也让我的背部和颈部不用拖着这块金属，给了我更大的工作屏幕，都提高了我的工作能力。在这个练习中，我希望你专注于你的工作领域，并确保在开始之前准备好：

- 你有足够的光线吗？你需要更少的光线吗？
- 你的椅子怎么样？你需要一个更好的键盘吗？
- 还有什么其他工具妨碍你了？你在 Windows 机器上尝试执行 Unix 的东西吗？试图在 Linux 上做 Mac 的东西？不要去买一台新电脑，但是如果你发现你想要做的事情太多，那么请考虑花一笔大的开销。
- 你的桌子怎么样？你有没有一个？你整天都在咖啡馆使用可怕的椅子和咖啡吗？
- 音乐怎么样？你听带歌词的音乐吗？我发现如果我听音乐没有歌词，我更容易专注于我的头脑中的声音，帮助我写作或编程。
- 你在开放式办公室工作，你的同事烦人吗？去买一双盖过耳朵的大耳机。当你戴着它们时，很明显你没有投入注意力，所以人们会离开你，比起你插上耳机而他们看不见，这样他们会觉得更温和一些。这也会阻止分心，并帮助你集中精力。

用这个练习来思考这样的主题，并试图简化和改善你的环境。有一件事 - 不要花很多钱买奇奇怪怪的工具。只需确定问题，然后尝试找到解决方法。

挑战练习

在这个挑战中，你要实现用于查找文件的 `find` 工具的基本版本。像这样运行 `find`：

```
find . -name "*.txt" -print
```

这将搜索当前目录中以 `.txt` 结尾的每个文件，并将其打印出来。 `find` 的命令行参数数量很多，所以你不应该在45分钟内实现它们。 `find` 的一般格式是：

- 开始搜索的目录： `.` 或 `/usr/local/` 。
- 一个过滤器参数，如 `-name` 或 `-type d` （目录类型的文件）。
- 对每个找到的文件执行的操作： `-print` 。

你可以执行有用的操作，例如在每个找到的文件上执行命令。如果要删除主目录中的每个Ruby文件，可以执行以下操作：

```
find . -name "*.rb" -exec rm {} \;
```

请不要运行它，除非意识到它会删除所有以 `.rb` 结尾的文件。 `-exec` 参数需要一个命令，将 `{}` 的任何实例替换为文件的名称，然后碰到 `;` （分号）时停止读取命令。在前面的命令中我们使用 `\;` ，因为 bash 和许多其他 shell 使用 `;` （分号）作为他们的语言的一部分，所以我们必须转义它。

这个练习将真正测试你使用 `argparse` 或 `sys.argv` 的能力。我建议你运行 `man` 找到参数列表，然后尝试使用 `find` 来确定你将要实现什么参数。你只有45分钟，所以你可能不会实现太多，但是 `-name` 和 `-type` ，以及 `-print` 和 `-exec` 是必不可少的。 `-exec` 参数将是一个挑战，所以将它保存到最后。

当你实现它的时候，尝试找到可以为你做的工作的库。你一定要查看 `subprocess` 模块和 `glob` 模块。同时要更仔细地查看 `os` 模块。

研究性学习

- 你实现了多少 `find` 的功能？
- 你发现用于改进这个实现的库是什么？
- 你将库的查找计算在45分钟内了吗？你可以说，开始 Hack 之前的研究并不算在内，这样做完全没有问题。如果你想要额外的挑战，那么就将你的研究包括在45分钟内。

深入学习

你可以在 45 分钟的 Hack 内实现多少 `find` 的功能？也许把它作为下一个星期的黑客热身挑战，来看看你能做些什么。记住，你应该试图拼凑出最好的、最丑陋的黑魔法。别担心，我不会告诉那些敏捷的人，你只是玩玩而已。

练习 7： `grep`

> 原文：Exercise 7: grep
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

`find` 命令在 45 分钟内应该可能是一个挑战，但它是一个很好的挑战。到了这个时间，你应该可以去掉尽可能多的，阻止你开始的障碍。你可能会发现，当你清除一些障碍时，你的技能会变得更糟。例如，我以前在开始工作之前先走路去喝咖啡。这花了我大约 30分钟，非常好，但 30 分钟多了会变成几个小时。我决定停止这样做，但是我的工作就费劲了。原来我还需要咖啡，所以我买了一台不错的咖啡机，学会了如何制作我自己的拿铁咖啡。现在我站了起来，给自己做一杯拿铁咖啡，然后去画一些画，这让我处于创意工作的模式。

你所做的一切不都是无效的，所以要小心不要因为占用时间而消除一些东西。有一些仪式和个人习惯，可以让你的大脑准备就绪。诀窍是不要消除这些，而是让它们在开始工作之前更容易做。

本书的第一部分中，您还应该了解时间管理的概念。设置45分钟的时间限制将使您非常清楚，当您不知道需要多久做某件事的时候。只有45分钟，你不能把 30 分钟浪费在调整你的 vim 窗口上，或者组织完美的目录结构，然后实现一个全新的排序算法。你必须节约你所实现的东西，以及命令你工作的东西。

处理项目的一个很好的方法是从最简单的事情开始，你可以首先配置并运行它。在 `find` 示例中，可以通过 `glob` 模块获取文件。具有较差时间管理技能的人，会立即尝试实现 `-exec` 参数，来证明它们是一个 NB 的程序员，但是 `-exec` 不能在没有-name的情况下工作，而且更难实现。决定的方法是告诉自己，你想要一些完成后才能使用的东西。如果45分钟之后，你可以使用 `-exec` ，但不能获取文件，那么你如何使用它？如果同一时间之后，你得到了一种方法，来列出匹配名称的文件，那么你已经完成了。

继续处理您的障碍列表，并评估您的开始怎么样，但现在开始看看时间管理。策略化你将要做的工作，以便如果你用完了时间，你做出来了一些可以用的东西。他们不必是完整的功能，但是两个可用的功能比 10 个不可用的更好，因为你忘记了他们所需要的最简单的东西。或者更糟的是 10 个不可用的东西，因为你实现了一半，然后就跳过去了，所以所有这些都不能用。

挑战练习

你现在要实现`grep`命令了。像往常一样，你应该去读的`grep`的`man`，然后玩转它。`grep`的目的是使用正则表达式在文件中搜索文本模式。你使用`glob`模块实现了`find`，这次的操作也一样，但在文件中而不是在目录中完成。例如，如果我想在我的书中搜索“help”这个词，我可以这样做：

```
grep help *.rst
```

`grep`的命令行参数相当简单。困难的部分是处理正则表达式，所以你应该依赖于`re`模块。该模块使您能够加载文件的内容，然后在里面搜索别人在命令行中提供给您的模式串。另一个提示是，您最有可能希望使用`readline`加载整个文件，而不是使用`read`。即使效率较低，`grep`的大部分选项都能更好地运行。

您可能还打算简单浏览练习 30，我在那里介绍正则表达式。

研究性学习

- `re`模块有什么特别的选项，使它更像`grep`吗？
- 您可以将您的`grep`黑魔法转换成您在`find`工具中使用的模块，来添加`grep`功能吗？

深入学习

`re`模块是非常重要的，所以花时间去真正研究它，并且学习你能学到的任何事情。我们将在本书的另一部分中使用它以及正则表达式。

练习 8： `cut`

> 原文：Exercise 8: cut
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

希望你正在深入学习 Python，甚至了解你自己和你的工作方式。在本书的这一部分，通过学习如何优化你的流程，你学到了流程和创造力的一些事情。的确，有阻碍的情况下你不能发挥创造力，但是你应该意识到，改善自己的个人流程的最简单方法是，观察你自己的工作。只做练习还不够。你需要留意你的个人工作方式，并努力改善它。

当你改进启动流程时，你可能会发现，需要几种不同的启动方法，来处理不同类型的项目。当我使用与这些命令行小工具类似的软件时，我可以从 Hack 代码来开始。当我需要使用 GUI 工作时，我发现我需要画出 UI，实现一个假版本，然后使其正常工作。当你继续阅读这本书的时候，你会学到两种工作方式并实践这个过程。

在这个练习中，我希望你专注于你的身体健康和行为。为了试图做它们的项目，程序员经常破坏他们的身体。工作感觉好像不应该对你造成伤害。你只是整天坐在桌子旁边，不需要砍伐树木，或者抓捕城市的罪犯。事实是，任何长时间作者，并且做有压力的事情的工作，可能破坏你的身体。为了避免这样，工作时要跟踪以下事情：

- 你的坐姿标准吗？挺直的姿势并不是很好，但是驼背也不是。让你的身体放松，并抬起头。
- 你把你的肩膀拉到你的耳朵吗？试着把它们放下来。
- 你绷紧手腕并把它们放在桌子上吗？尝试将它们悬在键盘上方，并保持它们不要太松，不要太紧。
- 你的头部在正前方并且是放松的吗？还是你把它拉紧，偏向一边来看另一台显示器？
- 你的椅子舒适吗？
- 你休息吗？45分钟是你停下来休息的最长时间。
- 你要去洗手间吗？我是认真的。如果你必须去，赶紧起来。最糟糕的是坐在那里拖着它。

还有更多，但这些主要的。我认为很多程序员觉得，如果他们离开他们的电脑，它们会消失甚至爆炸。电脑会耐心等待你返回，休息让你的大脑有机会以不同的方式处理这个问题。

你还应考虑打开电脑的网络摄像头并记录自己的工作。你可能会认为你不会懒散，但随后在激烈的战斗中，你会对你的身体做一些奇怪的事情，而不知道它。为你自己记录这个会话，然后寻找任何导致你的紧张，麻烦，背痛或只是奇怪的东西。

挑战练习

在这个练习中，你要实现`cut`工具。我真的很喜欢`cut`，因为它使我看起来像一个 Unix 术士，但是它真正做的是剪裁文本流。这是您可能会做的，最简单的小型文本处理工具，而且仍然实用。为了使用它，您需要另一个工具来为其提供一些文字，所以我们可以这样做：

```
ls -l | cut -d ' ' -f 5-7
```

这可能会向你提供乱码，但是在大多数系统上，它应该列出每个文件的用户名和组。`cut`命令接受一些选项，它设定类型的分隔符（`-d ' '`为一个空格字符），然后是要提取的字段列表（这里是`5-7`）。我们使用`ls -l`命令给它一些东西来剪裁。

这就完成了，所以阅读`man`的`cut`页面，看看你可以实现多少，同时检查你在工作时如何保持身体。

研究性学习

- Unicode 对您的实现有什么影响？

深入学习

记住你的身体是你的一部分，你的头脑是重要的想法是完全错误的。将你的身体看做无用的垃圾，只能使你的大脑工作效率较低，让你很长一段时间都不舒服。我建议您尽可能频繁地进行一些身体健康相关的活动。它可以是瑜伽，舞蹈，散步，远足或去健身房。任何保持身体健康的事情，让您的头脑无干扰地工作。

以这种方式思考：如果你的身体受到伤害，不断感觉不适，或者因不当使用而疲劳，那么你的大脑必须浪费周期来跟踪它并告诉你。如果你能保持身体良好，维护机器，那么你的大脑就不用担心它了。

最后，如果你的身体不像其它人那样好，那么尽力而为吧。没有人告诉你，你必须拥有我的身体来成为程序员。有关编码的重要事情之一，是任何人都可以做到它，即使他们的身体表现不能做许多其他事情。关键是不要让编程使你的情况更糟。请保持健康。

练习 9： `sed`

> 原文：Exercise 9: sed
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

使用这些小型项目来研究你自己是有用的，但让我们来看看你主要关注的主题：

- 开始工作的启动流程，例如你的文本编辑器，你可以打字打的多好，以及计算机内部发生的其他事情。
- 心理状态，当你开始工作时，建议将日志作为控制它的一种方式。
- 工作环境包括你的桌子，照明，椅子和你使用的电脑类型。
- 身体姿势和健康，以避免在工作时受伤。

在这个练习中，我们将采取这一改进计划，并进一步跟踪一些指标。你一直在使用小型命令行工具，阅读并确定其功能，然后花费45分钟来实现一个简单的黑魔法。现在你可以列举你的功能，确定优先级，然后确定在45分钟内可以完成的功能。事实上，你可以回溯目前为止所做的所有项目，以及你的更改笔记，并计算出这一指标，看看你是否在改进。

现在花点时间来回顾你的笔记，并大概估算每45分钟的完成的功能的百分比。在纸上绘制他们，然后看你的笔记，看是否有显著的变化，是好还是坏，当你改变你的工作方式的时候。然后在这个练习中，尝试根据你所做的改变，来预测你将做多少工作。你甚至可以尝试将一些障碍添加回你的流程，看看它如何影响你的生产力。

> 警告
>
> 请记住，这是个人指标，而不是与任何人分享的内容。这些几乎不科学，意味着你仅仅在分析你的工作方式中获得一些客观性。它们不是可以描述所有程序员的宏观指标，但是你最好相信，如果经理发现你有这些东西，他们会要求看到它们。然后，他们会要求你的团队中的每个人都开始做这些工作，然后管理层将会使用这些来引起大量的麻烦。将你的实验室笔记看做非常私人的记录，并且从不让任何人看到它。

挑战练习

这个练习将比其他练习更复杂，因为我们将要处理更多的正则表达式，并实现一个名为 `sed` 的工具。通过接受利用正则表达式替换模式串， `sed` 工具能够让你改变文本，然后在接收到的每一行中，确定要替换什么。难点可能在于实现 `sed` 的表达式格式，所以我建议你用两种方法来实现：

Lv1 具有命令行选项，用于最基本的 `sed` 用法，将一个字符串替换为另一个字符串。 Lv2 在这些命令行选项中启用正则表达式。 Lv3 是实现 `sed` 表达式格式。

使用 `sed` 的一个例子是，在文本流中将一个单词更改为另一个单词。如果我想更改 `ls` 的输出，以便使用“author”替换我的名字，我可以这样做：

```
ls -l | sed -e "s/zedshaw/author/g"
```

然而， `sed` 的实力在于使用正则表达式来匹配模式串并替换它们。如果你使用 vim 编辑器，那么你已经熟悉这种语法：

```
ls -l | sed -e"s / Jul [0-9] [0-9] / DATE / g"
```

你应该阅读 `man sed` 页面，但你可能需要进行更多的研究来实现它。我建议你晚上做你的研究，然后根据这项研究，第二天进行45分钟的 Hack。这将有助于你均衡你的指标，专注于你的工作。

研究性学习

- 当你制订指标时，你是否发现了异常或令人惊讶的事情？
- 在开始本课程之前，你的工作预测是什么？
- 它如何匹配你实际所做的事情？

深入学习

在这个练习的视频（付费）中，我会向你展示一个叫做“运行图”的东西。运行图是你希望监视的一些活动的简单图表，向你显示如何随时间变化。人们使用运行图来发现行为的巨大变化，因为它们是既简单又有效的可视化工具。你将在书中使用运行图，因为它们非常简单但非常强大。

练习 10： `sort`

> 原文：Exercise 10: sort
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你正在慢慢地构建我所说的个人流程实践（3P），这根本不是一个新的想法。3P 的目的是客观的洞察如何做事情，而避免杀死你的创造力和生产力。通过简单地跟踪小型指标和制作运行图来指导改进，你可以彻底改变你的工作状况。但是，这样做的风险在于，这会阻碍你快速入侵黑客或完成任务，或者你的 3P 的工作量将比你的实际工作更多。

在我的编程生涯中，我这样做了大约四年，并且它很好地让我认识到我自己和我的工作方式。它还切断了流程倡导者推动的许多谎言。我有一个简单的方法，来实际测试一些专家对程序设计的看法是否提高了我的个人生产力。我会说，我所做的唯一错误就是把它看得太重了，在四年时间里我的创造力被扼杀了。

这就是为什么你要在小型的快速 Hack 中，构建你的启动流程和工作环境的概念。只有 45 分钟的情况下，你没有时间收集复杂的指标，以及担心你怎么处理事情。稍后我们将专注于需要集中的实践，你将花费更多的时间并收集稍微更好的指标。当你工作的时候，尝试不让这些指标扼杀你的创造力，流程或快乐。如果你讨厌收集东西，那就不要这样做。找到一种自动化方式，或者提出另一个指标来代替。

对于这个练习，你正在制作完成功能的百分比的运行图表。这意味着在你工作之前，你必须列举在 `sort` 命令的所有功能，可以在它的标准手册页中可以找到，然后标记已完成的功能。记住要对它们进行排序，以便你可以完成足够的功能，以及该工具能够实际工作。对于不能实际工作的排序文本的工具，获得90%的分数意味着你实际完成了0%。

完成后，你应该制作每个项目的完成功能的百分比运行图，以便我们在下一个练习中进行分析。

挑战练习

在本练习中，你正在实现 `sort` 命令，这是一个非常简单的命令。它需要文本行，并按顺序对它们进行排序。它有相当多的有趣选项，所以你应该阅读 `man` 的 `sort` 页面，来弄清它可以做什么。大多数时候，人们只是使用 `sort` 来排序名称列表：

```
ls | sort
```

你也可以反向排序：

```
ls | sort -r
```

你也可以控制如何排序，例如忽略大小写：

```
ls | sort -f
```

或者，你甚至可以执行数值排序：

```
ls | sort -g
```

这对于 `ls` 的输出可能没有用，除非它们都是数字。

你的工作是尽可能实现这些功能，并跟踪你完成的每个功能。这些都应该在你的实验笔记中，以便以后分析。

研究性学习

- 你现在已经完成了一些改进的事情吗？尝试搜索并寻找其他人的流程建议。
- 我们是程序员和代码人。你试图找到能使你更有效率的代码吗？我的朋友 Audrey 和 Danny 有一个名为 cookie-cutter 的项目，你应该查看一下。
- 你现在应该研究如何计算一组数字的平均值。你将使用它在 Python 中计算运行图的中线。

深入学习

如果你真的想要一个正确的运行图，还需要计算你的数值的标准差。现在不需要，但如果你想要极其准确的，那么这是有帮助的。

练习 11： `uniq`

> 原文：Exercise 11: uniq
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在最后两个练习的开始，没有什么可说的了。你应该知道如何思考你的工作环境，你如何开始，你如何坐下来，影响你开始的任何事情。你也应该使用这些小小的 45 分钟的项目，突破了起始状态。如果你还没有弄清楚，设置一个 45 分钟的计时器，并大喊“来干个痛快！”，这是使自己开始的核心技巧。完成出色的工作的目标还没有完成，但是你已经起步了

你还应该有一个不错的实验笔记本，上面有运行图来描绘出你的改进效果如何。你的图表并不是非常科学，但是他们应该帮助你了解什么有用，什么没用。当你使用运行图时，你只需要在任一方向查找峰值，然后尝试为峰值找到“合理的原因”。如果峰值是有利的，那么试着找出原因并复现它。如果峰值是有害的，那么试着找出原因并防范于未然。

当我说“峰值”时，我的意思是重大变化。运行图应该是波动的。事实上，如果对于 45 分钟的 Hack，它保持不变，那也是坏的，你应该找出原因。正常流程在平均值周围波动和反弹，你应该只尝试找到任何一个方向的较大峰值的原因。如果你在以前的练习中进行了“深入学习”，则可以使用 `2 * std.dev` （两倍标准差）作为平均值上方和下方的线（俗称布林线），来发现问题。

> 注
>
> 运行图的更多演示请见此练习的视频（见原文）。它们在视频中更容易在视觉上解释。

挑战练习

`uniq` 命令仅仅接受 `sort` 产生的，排序后的行，并移除重复。当你想要获得列表的非重复行时，这非常方便。如果你已经实现了这些命令，那么你应该可以这样做：

```
history | sed -e "s/^[ 0-9]*//g" | cut -d ' ' -f 1 | sort | uniq
```

`history` 命令打印出你运行过的每个命令的列表。你的 `sed` 命令需要正则表达式，这将去掉 `history` 命令的头部。接下来我用 `cut` 来抓取第一个单词作为命令名称。之后，我通过 `uniq` 排序并执行它，并且你需要拥有所有你执行的命令。

实现 `uniq` 的足够功能，和任何其他必要的命令，使前面的命令工作。如果你的 `sed` 无法处理表达式，你可以更改格式，但在完成此练习后，你应该可以得到一个命令列表。

研究性学习

- 你现在有一个的命令列表，你可以开始执行它，如果你想深入学习的话。
- 这是第一个多项目练习，其中你将之前步骤中的练习组合成一个练习。你有没有找到关于你的流程的新东西？
- 你的运行图怎么样？他们有帮助吗？

深入学习

研究 Python 的图表库，看看你是否可以用 Python 生成这些运行图。你还应该开始跟踪你需要花费多长时间来开始，看看运行图是否可以帮助你缩短所需时间。

练习 12：复习

> 原文：Exercise 12: Review
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

我的疯狂方法的第一阶段对我来说是足够的，但不是为你准备的。我们现在要检查这本书这部分的策略，以便将来可以继续使用它。这个策略是：

- 你需要处理每个项目的起步。
- 为了隔离这个问题，你可以坐下来，在 45 分钟内完成一些小项目。这会把重点放在项目启动的问题位置，并让你重复该流程的一部分。
- 当你处理这些项目时，可以确定项目启动时的问题的可能原因。这可以表现为你的计算机设置，工作环境，心理思维过程或身体健康的形式。还有更多的，但是这是最可能的原因。
- 一旦确定了可能的原因，你就可以在 45 分钟 Hack 的小间隔内消除或改变它们。
- 最后，记录和绘制指标，看看这些可能的变化是否有帮助，但也要确保它们不会干扰你的表现。

这不需要是正式的科学流程，它就是实用的。所有你需要的是，将它当作一本日志来帮助你客观地看待你的工作方式。如果你做的正确，你会遇到你以前没想过的惊人的事情。收集数据迫使你探索新的可能性，并扩展一些东西，你认为它们可能是原因。

请记住，这个个人指标的日志不应与其他人分享，特别是管理人员。管理者会试图将这些指标强加于你，这是不可行的，如果这样做，那么你应该拒绝。这些是你的私人笔记，没有人有权阅读 -- 非常像日记或私人电子邮件。

挑战练习

最后的练习是让你选择最喜欢的工具，并花费一系列 45 分钟的时间，在一周或更多课程中优化它。使用你所学到的关于自己的一切，接受这个项目，从头开始，创造出更加健壮的东西。限制你自己 45 分钟一次，但不要把这个最终的项目当作一个黑魔法。相反，这是你正在开发的黑魔法的下一步。

在我完成一些快速的 Hack ，来测试一个想法之后，我将删除它或清理它。 如果这个黑魔法十分恶心，它就不能看到明天的太阳了，那么我就删除它，并以一个干净的开始重新实现它。 你不会忘记你所做的一切，并且必须解决这些问题，但是关注质量将会帮助你把它做得更干净。 如果黑魔法没有那么糟糕，那么我要做的就是在扩展之前进行清理。

将黑魔法转化为健壮的程序的一种有效技术是，使用自动化测试套件，将它的关键元素提取到库中。这迫使你将代码视为将在其他代码中使用的代码。我会这样做：

- 浏览文件，并将我的“黑魔法意识流”转换成一组函数。
- 然后我会使用 + DRY（不要重复你自己）重构代码，确保删除重复的代码，但不要太多。 零重复代码基本上是加密的随机的东西。
- 一旦清理和运行完毕，和之前一样，但是带有函数，我将这些函数放进模块，并确保原始代码保持有效。 记住，不要在清理过程中改变事情，只需要重新组织并修复它。
- 代码移动并再次正常工作之后，坐下来编写测试用例，确保我开始修改东西时，它能够在未来继续工作。

对于这个练习，你要选取你最喜欢的项目，并对它做这个“正式化”的过程。保持 45 分钟一次，并通过上述流程进行清理。当天工作时间最好超过 45 分钟，只需确保在每次会话之间休息 15-30 分钟。这是相同的时间框架，除非你不在 Hack，你应该认真对待。

研究性学习

- 将你的黑魔法代码与你的正式代码进行比较。你是否通过清理找到错误？是否还有其他改进？
- 如果黑魔法和清理后的代码在行为上几乎相同，那么你真的需要清理黑魔法吗？为什么你需要清理黑魔法，即使它的工作正常，并且可能更简单？
- 在你经常运行的命令列表中，尝试一个新的命令（参见练习 11），并尝试这个完整的流程。完成一个快速的 Hack，然后清理它，使它变得正式。

深入学习

这里是其它命令的列表，你应该尝试在 45 分钟内做出替代品：

- `ls`
- `rm`
- `rmdir`
- `mkdir`
- `cal`
- `tail`
- `yes`
- `false`

尝试实现它们中的一些吧。

第三部分：数据结构

> 原文：Part III: Data Structures
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你正在以你的方式构建个人流程，它让你以有限的阻碍快速起步。拥有良好的起步流程，以及培养一种尽管去做的能力，就是创造力的基础。创造力是一种流动性和放松的心态。如果你的起步充满阻碍和沮丧，那么很难进入这个流程。学习“点击”你的大脑，使其进入具有创造力的、松散的 Hack 模式，可以帮助你使用创造力解决问题，并提高生产力。

如果你做的是垃圾，那就没有意义了。首先，是的，显然，你所做的绝大多数都是垃圾，但你不想在你的余生中制造糟糕的软件。你需要平衡创造性的黑客心态和严谨的质量心态。我提倡人们在创造性表达和批判性思维模式之间切换。通过放松和创造力，你想出你的想法并实现它，然后通过批评自己的工作来使他们可靠和品质高。

在第二部分中，当你跟踪 45 分钟内实现的功能数，并寻找可以改进你的启动流程的地方时，实际上就做到了它。但是，由于批判性思维模式是创造力的杀手，因此你无法同时 Hack 和分析你的流程。这个建议几乎涵盖了我所知道的每一个创造性规律，并帮助你在工作时不使用自己的方式。

> 注
>
> 创造过程中的批评会扼杀你的想象力。没有批评的创造只会产生垃圾。你需要这两个，但不是同时。

在第三部分中，你将切换到专注于质量和开发个人流程，从而提高你的质量。为了使其变得简单，我只会将质量定义为：

> 低缺陷率和可理解的代码。

大多数程序员在这两个方面绝对是糟糕的。绝大多数开发人员认为，当编译完成时，他们的工作就完成了，就是这样。他们运行了测试套件，所以就完成了！我称这个“程序员风格的完成”，其中它们对自己的作品没有自我批评的评价，因为他们完全相信他们的电脑来找出所有的缺陷。他们似乎从来不在乎，别人是否可以理解他们的代码，只关注它是否运行良好来满足最低限度。如果你曾问过他们每天的缺陷率，他们会瞪着你，说这并不重要。代码覆盖？呸。他们的测试套件有 10 万行代码！它肯定测试了一切东西！

为了成为一名更好的程序员，你必须开始开始做一些残酷的事情，观察自己的质量指标和实践。我说这项工作是残酷的，因为它清晰并明确地展示出你是多么糟糕，对于那些开心地认为他们很棒的人来说，这可能是个悲剧。那些具有骗子综合症的人，会发现这种质量分析令人耳目一新，因为它会给你一个合理的想法，你的工作有多好，以及一个改进计划。

通过数据结构测试质量

数据结构是一个简单的概念。你的计算机拥有内存和放入内存的数据。你可以随意填充它，也可以提供一种使数据更容易处理的结构。自从“计算机科学”开始以来，人们一直在分析如何为不同目的构建数据，然后这些结构有多好。由于数据结构定义好了，我们可以使用它们来研究你的质量实践。你将实现每个数据结构并进行测试，然后通过两个步骤来确定实现的质量。

你进行数据结构练习的流程如下：

- 每个练习都将描述数据结构，以及你可以做什么。这个描述是中文，图表和示例代码。我将给出一个没有代码的结构的完整描述，因为你需要实现它，并使其正确。
- 你可能还需要一套必须通过的测试，但是这些测试也可能用文本写成，因此你还将编写自动测试。
- 你将在 45 分钟内持续进行训练来做一些东西，然后休息一阵子，但你可以在每次实现中花费更多时间。我建议你做一些简单的黑魔法，然后“认真起来”，并在更多的时间段中优化你的实现。
- 当你相信你“完成”时，你会切换到批评模式，并开始了解你的实际情况。你将遵循一个审计流程，让你仔细阅读你的代码，并查找错误，跟踪你的代码。
- 最后，你将修复在审计阶段发现的缺陷，并继续处理这个练习，直到完成。

这就是相关的流程，所以这部分的前两个练习（练习 13 和 14）将由我完成，现场制作，所有的缺陷都是我做的，所有的代码都是我写的。你可以在视频中看到这个流程的工作原理，并在练习中阅读我的代码，以便你了解预期的东西。我会遵循我上面所述的严格流程，尽可能接近，所以你需要仔细观看视频。

如何学习数据结构

有一种正式的数学方法，用于研究算法和数据结构，但我不会设计过多背后的理论。如果你对这个微小的简介感兴趣，那么你可以阅读几本这个主题的书，并花几年研究这个计算机科学分支。在这本书中，我将向你提供练习，以便你学习如何从记忆中实现它们，并了解它们的工作原理。你不需要正式的证明，只需要简单的 Python 代码和反复尝试。

通过这些练习，我希望你遵循一种具体的方式来研究它们，以便你可以从记忆中实现它们。当我学习音乐，和尝试画出我看到的东西时，我使用这个相同的流程。它适用于任何东西，其中你需要记住一个概念，但也可以通过创造力应用于不同的情况，所以你不能只是记住

它。相反，你执行我所说的“记忆，尝试，检查”：

- 构建一切信息和材料，它描述你必须记住的所有事情。尽你所能来记忆并记住它，即使它只是信息的一小部分。
- 把所有的信息拿走，所以你看不到它。我喜欢把它放在不同的房间，所以如果我需要再次查找，我必须离开我的作品。
- 尝试从记忆中创建所需的东西。尝试放下任何东西，无论是对的或错的。
- 当你用尽了你记住的东西后，把你所做的一切拿过来，然后返回你的信息并进行比较。标记所有你做错了的东西，然后再试一次。
- 使用你的错误列表，专注于记忆，以便你在下次尝试时更正错误，并重新做一遍。

我喜欢进行 2~15 分钟的记忆，然后进行 10~45 分钟的尝试，但你会知道你什么时候用完所有知识，需要去获得更多。我将给出一个具体的例子，解释我如何根据我的记忆来画画：

- 我要画一朵花，所以我把花放在我房子的一个房间里，我的画在另一个房间里。
- 我坐在花的房间里，盯着花。我画出那朵花。我用手指跟踪它，试着在我的脑海里想象它。我想象自己画每个花瓣，茎，和一切东西。我记得比例。我甚至可以使用笔记记录颜色，并尝试在花的房间中混合颜色。
- 我把所有东西都放在花的房间里。我很快回到画室，试着唤醒花的记忆，找出下一步要画的东西。也许叶子是我最终熟悉的东西。我画出它。也许现在花盆很清楚，我画了一些。我继续闭上眼睛，试图想起图像，然后尝试画出来。
- 当我被卡住或者我用完时间的时候，我站起来，把我的小画板带入花的房子，并将其与我看到的比较。然后我会记录我的错误。一只花瓣太长吗？花盆的角度错了吗？土壤太暗了吗？我记下笔记，弄清楚我错了什么。
- 然后我把画带回画室，并回到花的房间，在下一个回合中，用这个错误列表继续从记忆中学习。

我从这个流程中画出的画，通常相当奇怪，但接近于原件，这取决于我用了多少回合，而且我多长时间练习一次。最终这帮助我变得更好，并快速捕捉到我看到的东西，因为我可以在我的记忆中将更多的视觉信息保存更长时间。

当你进行这些算法练习时，你可以使用相同的流程，来发展你在面试中根据需要回忆它们的能力。你应该首先坐下来，使用你可以使用的所有信息来实现它们，并了解它们的工作原理。记住你不明白的东西很难记住。你有一个好的实现后，你可以开始训练你的记忆。

- 将所有书籍，笔记，图表和关于该算法的信息放在一个房间中，将计算机放在另一个房间中。如果需要，打印出你的代码。
- 花费 15 分钟的时间来学习算法房间中的信息，记下笔记，绘制更多的图表，可视化数据的流动方式，并且做任何其他可以想到的事情来学习。
- 将所有信息留在算法房间，走进笔记本电脑的房间，坐下来尝试从记忆中实现它们。在检查你的作品之前，不要花费超过 45 分钟。
- 带着你的笔记本电脑进入算法房间，并记录你的错误。
- 把你的笔记本电脑放回去，然后再回到算法房间，再进行一遍记忆和学习。专注于所有

你做错了的事情，这将使它更容易。

最初的几次中，这样做会令人沮丧，但很快，你会发现它变得更加容易，而且在许多情况下，你可以冥想来使其生效。

练习 13：单链表

> 原文：Exercise 13: Single Linked Lists
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你将实现的第一个数据结构是单链表。我将描述数据结构，列出你应该实现的所有操作，并给你实现需要通过的单个测试。你应该首先尝试使用此数据结构，然后再观看我的实现和审计视频，以便你了解该过程。

> 警告
>
> 这些都不是数据结构的高效实现。它们故意做成朴素和缓慢的，以便我们可以在练习 18 和 19 中讲解度量和优化。如果你在行业工作中尝试使用这些数据结构，就会有性能问题。

描述

在面向对象语言（如 Python）中处理许多数据结构时，你需要理解三个常见概念：

- “节点”，通常是数据结构的容器或存储单元。你的值保存在这里。
- “边”，但我们会叫它“指针”或“链接”，它指向其他节点。这些都放在每个节点内，通常作为实例变量。
- “控制器”，它是一些类，知道如何使用节点中的指针来正确构造数据。

在 Python 中，我们将映射这些概念，如下所示：

- 节点只是一个类定义的对象。
- 指针（边）只是节点对象中的实例变量。
- 控制器是另一个简单的类，它使用节点存储所有内容并构建数据。这是所有的操作（`push`，`pop`，`list` 等）的地方，通常控制器的使用者从来没有真正处理节点或指针。

在一些关于算法的书中，你将看到这样的实现，将节点和控制器组合成一个类，但这是非常混乱的，也违反了设计中的问题分离。最好将节点与控制类分开，以便只做一件事并且把它做好，以及你知道错误在哪里。

想象一下，我们想要存储一系列汽车。我们有第一辆车，后面是第二辆，直到最后一辆。想象这个列表，我们可以开始设想一个节点/指针/控制器设计：

- 节点包含每个车的描述。也许这只是一个 `Car` 类的 `node.value` 变量。如果你很懒，我们可以调用这个 `SingleLinkedListNode` 或 `SLLNode` 。
- 然后，每个 `SLLNode` 具有一个链接，指向链表中下一个节点。访问 `node.next` 可以让你访问下一辆车。
- 控制器，简单地称为 `SingleLinkedList` ，具有诸如 `push` ， `pop` ， `first` 或 `count` 之类的操作，它们接受 `Car` ，并且使用节点在内部进行存储。当你将汽车 `push` 到 `SingleLinkedList` 控制器上时，它将处理在一个节点的内部链表，来将其存储在最后。

> 注
>
> 当 Python 有个相当好用并且快速的 `list` 时，为什么我们要这么做呢？完全是为了学习数据结构。在真实世界中，你可以使用 Python 的 `list` 并继续。

为了实现 `SingleLinkedListNode` ，我们需要一个简单的类，如下：

```python
class SingleLinkedListNode(object):

    def __init__(self, value, nxt, prev):
        self.value = value
        self.next = nxt

    def __repr__(self):
        nval = self.next and self.next.value or None
        return f"[{self.value}:{repr(nval)}]"
```

我们必须使用单词 `nxt` ，因为 `next` 是 Python 中的保留字。除此之外，这是一个非常简单的课程。最复杂的是 `__repr__` 函数。当你使用 `%r` 格式或在节点上调用 `repr()` 时，这会打印调试输出。它应该返回一个字符串。

> 注
>
> 现在花时间了解如何使用 `SingleLinkedListNode` 类手动构建列表，然后手动遍历它。这是一个很好的45分钟 hack spike，尝试练习它。

控制器

一旦我们在 `SingleLinkedListNode` 类中定义了我们的节点，我们可以确切地知道控制器应该做什么。每个数据结构都有所需的常用操作列表，使其有用。不同的操作花费不同的内存（空间）和时间，一些是昂贵的，另一些是快速的。 `SingleLinkedListNode` 的结构使得一些操作非常快，但是许多其他操作非常慢。在实现过程中，你将会了解到它。

查看操作的最简单方法是，查看 `SingleLinkedList` 类的框架版本：

```python
class SingleLinkedList(object):

    def __init__(self):
        self.begin = None
        self.end = None

    def push(self, obj):
        """将新的值附加到链表尾部。"""

    def pop(self):
        """移除最后一个元素并返回它。"""

    def shift(self, obj):
        """将新的值附加到链表头部。"""

    def unshift(self):
        """移除第一个元素并返回它。"""

    def remove(self, obj):
        """寻找匹配的元素并从中移除。"""

    def first(self):
        """返回第一个元素的*引用*，不要移除。"""

    def last(self):
        """返回最后一个元素的*引用*，不要移除。"""

    def count(self):
        """计算链表中的元素数量。"""

    def get(self, index):
        """获取下标处的值。"""

    def dump(self, mark):
        """转储链表内容的调试函数。"""
```

在其他练习中，我只会告诉你这些操作，并留给你来弄清楚，但是对于这个练习，我会指导你实现。查看 `SingleLinkedList` 中的函数列表，来查看每个操作以及如何使用的注释。

测试

我现在要向你提供测试，实现这个类时，你必须使其能够工作。你会看到我已经遍历了每一个操作，并试图覆盖大部分的边界情况，但是当我进行审计时，你会发现实际上我可能错过了一些。人们常常不会对一些案例进行测试，例如“零个元素”和“一个元素”。

```python
from sllist import *

def test_push():
    colors = SingleLinkedList()
    colors.push("Pthalo Blue")
    assert colors.count() == 1
    colors.push("Ultramarine Blue")
    assert colors.count() == 2

def test_pop():
    colors = SingleLinkedList()
    colors.push("Magenta")
    colors.push("Alizarin")
    assert colors.pop() == "Alizarin"
    assert colors.pop() == "Magenta"
    assert colors.pop() == None
```

```python
def test_unshift():
    colors = SingleLinkedList()
    colors.push("Viridian")
    colors.push("Sap Green")
    colors.push("Van Dyke")
    assert colors.unshift() == "Viridian"
    assert colors.unshift() == "Sap Green"
    assert colors.unshift() == "Van Dyke"
    assert colors.unshift() == None

def test_shift():
    colors = SingleLinkedList()
    colors.shift("Cadmium Orange")
    assert colors.count() == 1

    colors.shift("Carbazole Violet")
    assert colors.count() == 2

    assert colors.pop() == "Cadmium Orange"
    assert colors.count() == 1
    assert colors.pop() == "Carbazole Violet"
    assert colors.count() == 0

def test_remove():
    colors = SingleLinkedList()
    colors.push("Cobalt")
    colors.push("Zinc White")
    colors.push("Nickle Yellow")
    colors.push("Perinone")
    assert colors.remove("Cobalt") == 0
    colors.dump("before perinone")
    assert colors.remove("Perinone") == 2
    colors.dump("after perinone")
    assert colors.remove("Nickle Yellow") == 1
    assert colors.remove("Zinc White") == 0

def test_first():
    colors = SingleLinkedList()
    colors.push("Cadmium Red Light")
    assert colors.first() == "Cadmium Red Light"
    colors.push("Hansa Yellow")
    assert colors.first() == "Cadmium Red Light"
    colors.shift("Pthalo Green")
    assert colors.first() == "Pthalo Green"

def test_last():
    colors = SingleLinkedList()
    colors.push("Cadmium Red Light")
    assert colors.last() == "Cadmium Red Light"
    colors.push("Hansa Yellow")
    assert colors.last() == "Hansa Yellow"
    colors.shift("Pthalo Green")
    assert colors.last() == "Hansa Yellow"

def test_get():
    colors = SingleLinkedList()
    colors.push("Vermillion")
    assert colors.get(0) == "Vermillion"
    colors.push("Sap Green")
    assert colors.get(0) == "Vermillion"
    assert colors.get(1) == "Sap Green"
    colors.push("Cadmium Yellow Light")
    assert colors.get(0) == "Vermillion"
    assert colors.get(1) == "Sap Green"
    assert colors.get(2) == "Cadmium Yellow Light"
    assert colors.pop() == "Cadmium Yellow Light"
    assert colors.get(0) == "Vermillion"
    assert colors.get(1) == "Sap Green"
    assert colors.get(2) == None
    colors.pop()
    assert colors.get(0) == "Vermillion"
    colors.pop()
```

```
    assert colors.get(0) == None
```

仔细研究此测试，以便你在尝试实现之前，先了解每个操作应如何工作。我不会一次将所有这些代码写入文件。相反，最好每次只做一个测试，并使其小部分能够工作。

> 注
>
> 这里，如果你不熟悉自动化测试，你可能想要观看视频，来看我怎么做。

审计入门

当你执行每个测试时，你将审计代码来找到缺陷。最终，你将跟踪你在审计中找到的缺陷数量，但现在你需要在写完代码之后执行审计。“审计”类似于政府认为你偷税漏税的时候，税务局所做的工作。他们遍历每笔交易，每笔收入金额，所有支出金额，以及你为什么这样来花费。代码审核与之类似，因为你遍历每个函数，并分析所有输入参数，以及所有输出值。

要进行基本的审计，你将执行此操作：

- 从你的测试用例开始。在这个例子中我们来审计 `test_push` 。
- 查看第一行代码，并确定正在调用什么以及正在创建什么。在这种情况下，它的 `colors = SingleLinkeList()` 。这意味着我们正在创建 `colors` 变量，并调用 `SingleLinkeList.__init__` 函数。
- 跳到 `__init__` 函数的顶部，保持测试用例和目标函数（ `__init__` ）并排。确认你已经这样做了。然后，确认你使用数值和类型正确的函数参数来调用它。在这种情况下 `__init__` 只需要 `self` ，它应该是正确的类型。
- 然后进入 `__init__` 并逐行审计，以相同的方式确认每个函数调用和变量。它的参数数量正确吗？类型正确吗？
- 在每个分支（ `if` 语句， `for` 循环， `while` 循环）中，确认逻辑是正确的，并且它处理逻辑中的任何可能的条件。 `if` 语句的 `else` 子句有错误吗？循环能结束吗？然后潜入每个分支，以相同方式跟踪函数，潜入，检查变量，回来，并检查返回值。
- 当你到达一个函数结尾或任何 `return` 的时候，跳回到 `test_push` 调用者，来检查返回值是否匹配期望值，当你调用它的时候。记住，尽管如此，你也可以对 `__init__` 中的每个调用搞这么做。
- 最后，当你到达 `test_push` 函数的末尾时，你就完成了，并且已经完成了它调用的每个函数的递归检查。

这个流程一开始似乎很乏味，是的，但是你会越来越快，在视频中你会看到，在运行每个测试之前我都这么做（或至少我真的努力尝试这么做）。我按照以下流程：

- 写一些测试代码。
- 编写代码使测试工作。
- 审计二者。
- 运行测试，看看我是否正确。

挑战练习

我们现在到达了这个部分，你已经准备好尝试它了。首先，浏览测试并研究它的作用，并研究 `sllist.py` 中的代码，来弄清楚你需要做什么。我建议当你尝试在 `SingleLinkeList` 中实现一个函数时，首先写一些注释来描述它做了什么，然后填充 Python 代码来使这些注释工作。你会看到我在视频中这样做。

当你花了一两个 45 分钟的会话来 Hack 它并试图让它工作时，现在是观看视频的时候了。你首先需要尝试它，以便更好地了解我正在尝试的事情，这样可以使视频更容易理解。视频中我只是编程而不说话，但我会做一个旁白来讨论发生了什么。视频也更快来节省时间，我会剪切掉任何无聊的错误或时间的浪费。

一旦你看到我是怎么做的，你已经做了笔记（对吗？），然后去尝试更严格的东西，并尽可能仔细地执行代码审核过程。

审计

编写代码后，请确保执行第三部分中描述的审计流程。如果你不太确定如何完成，我也将在视频中为这个练习执行审计。

深入学习

为这次练习准备的深入学习是，完全根据我在第三部分的介绍中描述的方式，尝试再次实现该算法。你还应该尝试思考，这个数据结构中的哪些操作最有可能很慢。完成后，对你创建的内容执行审计。

练习 14：双链表

> 原文：Exercise 14: Double Linked Lists
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

以前的练习可能需要花一段时间才能完成，因为你必须弄清楚如何使单个链表工作。希望视频为你提供完成练习的足够信息，并向你展示如何审计代码。在本练习中，你将实现更好的链表 `DoubleLinkedList` 。

在 `SingleLinkedList` 中，你应该已经意识到，涉及列表末尾的任何操作，都必须遍历每个节点，直到到达末尾。 `SingleLinkedList` 仅仅对于列表前面是高效的，那里你可以轻松地更改 `next` 指针。 `shift` 和 `unshift` 操作非常快，但 `pop` 和 `push` 的开销随链表增大而增大。你可以通过保留下一个元素到最后一个元素的引用来加速，但是如果要替换该元素，该怎么办？同样，你必须遍历所有的元素来找到这个元素。你可以通过细微变化来获得一些速度改进，但更好的解决方案是，修改结构，使其可以从任何位置工作。

`DoubleLinkedList` 与 `SingleLinkedList` 几乎一样，但它还有 `prev` （上一个）链接，指向它前面的 `DoubleLinkedListNode` 。每个节点有一个额外的指针，使许多操作突然变得容易得多。你还可以在 `DoubleLinkedList` 中，轻易添加一个指向 `end` 的指针，所以你可以直接访问头部和尾部。这使得 `push` 和 `pop` 效率更加高效，因为你可以直接访问尾部，并使用 `node.prev` 指针获取上一个节点。

考虑到这些变化，我们的节点类看起来像这样：

```
class DoubleLinkedListNode(object):

    def __init__(self, value, nxt, prev):
        self.value = value
        self.next = nxt
        self.prev = prev

    def __repr__(self):
        nval = self.next and self.next.value or None
        pval = self.prev and self.prev.value or None
        return f"[{self.value}, {repr(nval)}, {repr(pval)}]"
```

所有添加的东西就是 `self.prev = prev` ，以及在 `__repr__` 中处理它。 `DoubleLinkedList` 类的实现使用 `SingleLinkedList` 的相同方式，除了你需要为链表末尾添加一个额外的变量。

```
class DoubleLinkedList(object):

    def __init__(self):
        self.begin = None
        self.end = None
```

引入不变条件

所有要实现的操作都一样，但是我们有一些额外的事情需要考虑：

```
def push(self, obj):
        """将新的值附加到链表尾部。"""

    def pop(self):
        """移除最后一个元素并返回它。"""

    def shift(self, obj):
        """将新的值附加到链表头部。"""

    def unshift(self):
        """移除第一个元素并返回它。"""

    def detach_node(self, node):
        """你有时需要这个操作，但是多数都在 remove() 里面。它应该接受一个节点，将其从链表分离，无论节点是否在头部、尾部还是在中间。"""

    def remove(self, obj):
        """寻找匹配的元素并从中移除。"""

    def first(self):
        """返回第一个元素的*引用*，不要移除。"""

    def last(self):
        """返回最后一个元素的*引用*，不要移除。"""

    def count(self):
        """计算链表中的元素数量。"""

    def get(self, index):
        """获取下标处的值。"""

    def dump(self, mark):
        """转储链表内容的调试函数。"""
```

使用 `self.end` 指针，你现在必须在每个操作中处理更多的条件：

- 是否有零个元素？那么 `self.begin` 和 `self.end` 都需要是 `None` 。
- 如果有一个元素，那么 `self.begin` 和 `self.end` 必须相等（指向同一个节点）。
- 第一个节点的 `prev` 必须始终为 `None` 。
- 最后一个节点的 `next` 必须始终为 `None` 。

这些事实必须在 `DoubleLinkedList` 的生命周期中维持，这使得它们成为“不变条件”或者只是“不变量”。不变量的想法是，无论如何，这些基础检查显示了结构正常工作。查看不变量的一种方法是，任何重复调用的测试或者 `assert` 调用可以移动进一个函数，叫做 `_invariant` ，它执行这些检查。然后，你可以在测试中或每个函数的开始和结束处调用此函数。这样做会减少你的缺陷率，因为你假设“不管我做什么，这些都是真的”。

不变量检查的唯一问题是它们的运行花费时间。如果每个函数调用也调用另一个函数两次，那么你就为每个函数增加了潜在的重要负担。如果你的 `_invariant` 函数也会导致成本增加，就变得更糟。想象一下，如果你添加了不变量："所有节点都有一个 `next` 和 `prev` ，除了第一个和最后一个。这意味着每个函数调用都遍历列表两次。当你必须确保类一直有效时，这是值得的。如果不是，那就是一个问题。

在这本书中，你可以使用 `_invariant` 函数，但请记住，你不需要始终使用它们。寻找方法，只在测试套件或调试中激活它们，或者在初始开发过程中使用它们，这是有效使用它们的关键。我建议你只在函数顶部调用 `_invariant` ，或者只在测试套件中调用它们。这是一个很好的权衡。

挑战练习

在本练习中，你将实现 `DoubleLinkedList` 的操作，但此时你还将使用 `_invariant` 函数来检查每个操作之前和之后是否正常。最好的方法是,在每个函数的顶部调用 `_invariant` ，然后在测试套件中的关键点调用它。 `DoubleLinkedList` 的测试套件几乎是 `SingleLinkedList` 测试的复制粘贴副本，除了在关键点添加 `_invariant` 调用。

与 `SingleLinkedList` 一样，你需要自己手动研究此数据结构。你应该在纸张上绘制节点结构，并手动执行某些操作。接下来，在 `dllist.py` 文件中手动实现 `DoubleLinkedListNode` 。之后，花费一两个 45 分钟的时间，来尝试黑掉一些操作来弄清楚。我推荐 `push` 和 `pop` 。之后，你可以观看视频以查看我的工作，以及如何组合使用我的代码的审计和 `_invariant` 函数，来检查我在做什么。

深入学习

与以前的练习一样，你要按照记忆再次实现此数据结构。把你所知道的东西放在一个房间里，你的笔记本电脑在另一个房间。你将要执行此操作，直到你可以按照记忆实现 `DoubleLinkedList` ，而无需任何参考。

练习 15：栈和队列

> 原文：Exercise 15: Stacks and Queues
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

当处理数据结构时，你将经常遇到类似于另一种结构的结构。 `Stack` 类似于练习13中的 `SingleLinkedList` ，以及 `Queue` 类似于练习14中的 `DoubleLinkedList` ，唯一区别是 `Stack` 和 `Queue` 限制了可能的操作，以简化它们的使用方式。这有助于减少缺陷，因为你不能意外地像 `Queue` 那样使用 `Stack` 并导致问题。在 `Stack` 中，节点被“压入”“栈顶”，然后从顶部“弹出”。在队列中，节点压入“尾部”，之后从“头部”弹出。这些操作都是 `SingleLinkedList` 和 `DoubleLinkedList` 的简化，其中 `Stack` 只允许 `push` 和 `pop` 操作， `Queue` 只允许 `shift` 和 `unshift` 。

> 译者注：实际上是 `push` 和 `unshift` 。

当可视化堆栈时，你应该想到你的地板上的一堆书。想像我在书架上的那种很重的艺术书，如果我堆叠了20个，可能会重约100磅。当你为这些书构建栈的时候，你不能抬起整个栈，并且把书放在底部，对吧？不，你把书放在栈的顶部。你把它放在那儿，但我们也可以使用“推”描述这个动作。如果你想从栈中获取一本书，你可能会抬起一些书，然后抓住一本书，但是最终你可能要从顶部拿出一些书，才能获取底部得数。你可以从顶部抬起每本书，或者在我们的例子中，我们会说“从顶部弹出一本书”。

如果你想像在银行排队，队列有“头部”和“尾部”，可视化队列是最简单的。通常有一个绳索迷宫，它的末尾有一个入口，出口处是检票员。你可以通过进入这条绳索迷宫的“尾部”进入队列，我们称之为 `shift` ，因为这是 `Queue` 数据结构中的常见编程属于。一旦你进入银行（队列），你不能越过等候线然后离开，否则其余的人会生气。所以你必须等待，随着你前面的每个人都离开了等候线（对你而言是 `unshift` ），你离“头部”更近了。一旦你达到了头部，那么你可以退出，我们称之为 `unshift` 。

很多时候，你可以找到数据结构的真实世界示例，来帮助你可视化其工作原理。你现在应该花点时间来绘制这些场景，或者实际上得到书籍的栈并测试这些操作。你可以找到与 `Stack` 和 `Queue` 类似的其他真实情况吗？

挑战练习

我现在打算让你做一个基于代码的挑战练习，并且从它们的描述中实现数据结构。在这个挑战中，你首先需要使用这里的起始代码，以及你从练习 13 中了解的 `SingleLinkedList`，实现 `Stack` 数据结构。完成之后，你将尝试从零开始实现 `Queue` 数据结构。

`StackNode` 节点类几乎和 `SingleLinkedListNode` 相同，而事实上我只是复制过来并更名：

```python
class StackNode(object):

    def __init__(self, value, nxt):
        self.value = value
        self.next = nxt

    def __repr__(self):
        nval = self.next and self.next.value or None
        return f"[{self.value}:{repr(nval)}]"
```

`Stack` 控制类和 `SingleLinkedList` 十分类似，除了我使用 `top` 代替了 `first`。这样匹配 `Stack` 的概念。

```python
class Stack(object):

    def __init__(self):
        self.top = None

    def push(self, obj):
        """Pushes a new value to the top of the stack."""

    def pop(self):
        """Pops the value that is currently on the top of the stack."""

    def top(self):
        """Returns a *reference* to the first item, does not remove."""

    def count(self):
        """Counts the number of elements in the stack."""

    def dump(self, mark="----"):
        """Debugging function that dumps the contents of the stack."""
```

现在你的挑战是实现 `Stack`，并为其执行测试，类似于在练习 13 中进行的测试。请确保你的测试涵盖了每一个操作，你可以以任何方式。记住，尽管如此，堆栈的 `push` 操作必须在顶部，所以有到顶部的链接。

一旦你使 `Stack` 正常工作，你应该实现 `Queue`，但它基于 `DoubleLinkedList`。（译者注：其实单链表也行，因为只有尾部弹出的操作比较困难。你可以在尾部插入，在头部弹出。）`Stack` 中的内容应该与 `SingleLinkedList` 基本内部结构相同，只需更改允许的功能。`Queue` 也一样。花点时间来绘制队列的工作原理，然后弄清楚它如何限制 `DoubleLinkedList`。一旦你完成了，创建你的队列。

破坏它

破坏这些数据结构仅仅是不要维持约束。看看如果一个操作无法使用正确的尾部会发生什么。

你可能还注意到，它有“偏移一位”的持久性错误。在我的设计中，当结构为空时，我设置了`self.top = None`。这意味着当你达到 0 个元素时，你必须对`self.top`做一些特殊处理。一个替代方法是使`self.top`总是指向一个`StackNode`（伪造的头节点），并假设当你有这个最后的元素时，结构是空的。尝试它，看看它如何改变你的实现。这样会更容易出错还是更不容易出错？

深入学习

这些数据结构有很多操作是非常低效的。回顾你为每个数据结构编写的代码，并尝试猜测哪些函数最慢。一旦你有了想法，尝试解释为什么他们可能很慢。研究其他人对这些数据结构的看法。在练习 18 和 19 中，你将学习对这些数据结构进行一些性能分析并进行调整。

最后，你真的需要实现一个全新的数据结构吗，还是简单地“包装”`SingleLinkedList`和`DoubleLinkedList`数据结构？这如何改变你的设计？

练习 16：冒泡、快速和归并排序

> 原文：Exercise 16: Bubble, Quick, and Merge Sort
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你现在将尝试为你的 `DoubleLinkedList` 数据结构实现排序算法。对于这些描述，我将使用“数字列表”来表示随机的事物列表。这可能是一堆扑克牌，一张纸上的数字，名称列表或其他任何可以排序的东西。当你尝试排序数字列表时，通常有三个备选方案：

> 冒泡排序
>
> 如果你对排序一无所知，这是你最可能尝试的方式。它仅仅涉及遍历列表，并交换你找到的任何乱序偶对。你不断遍历列表，交换偶对，直到你没有交换任何东西。很容易理解，但是特别慢。
>
> 归并排序
>
> 这种排序算法将列表分成两半，然后是四个部分，直到它不能再分割为止。然后，它将这些返回的东西合并，但是在合并它时，通过检查每个部分的顺序，以正确的顺序进行操作。这是一个聪明的算法，在链表上工作得很好，但在固定大小的数组上并不是很好，因为你需要某种 `Queue` 来跟踪部分。
>
> 快速排序
>
> 这类似于归并排序，因为它是一种“分治”算法，但它的原理是交换分割点周围的元素，而不是将列表拆分合并在一起。在最简单的形式中，你可以选择从下界到上界的范围和分割点。然后，交换分割点上方的大于它的元素，和下方的小于它的它元素。然后你选择一个新的下界，上界和分割点，它们在这个新的无序列表里面，再执行一次。它将列表分成更小的块，但它不会像归并排序一样拆分它们。

挑战练习

本练习的目的是，学习如何基于“伪代码”描述或“p-code”的实现算法。你将使用我告诉你的参考文献（主要是维基百科）研究算法，然后使用伪代码实现它们。在这个练习的视频中，我会在这里快速完成前两个，更细节的东西留作练习。那么你的工作就是自己实现快速排序算法。首先，我们查看维基百科中冒泡排序的描述，来开始：

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* 如果这个偶对是乱序的 */
            if A[i-1] > A[i] then
                /* 交换它们并且记住 */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
```

你会发现，因为伪代码只是对算法的松散描述，它最终在不同书籍，作者和维基百科的页面之间截然不同。它假设你可以阅读这种“类编程语言”，并将其翻译成你想要的内容。有时这种语言看起来像是一种叫做 Algol 的旧语言，其他的时候它会像格式不正确的 JavaScript 或者 Python 一样。你只需要尝试猜测它的意思，然后将其翻译成你需要的。这是我对这个特定的伪代码的最初实现：

```python
def bubble_sort(numbers):
    """Sorts a list of numbers using bubble sort."""
    while True:
        # 最开始假设它是有序的
        is_sorted = True
        # 一次比较两个，跳过头部
        node = numbers.begin.next
        while node:
            # 遍历并将当前节点与上一个比较
            if node.prev.value > node.value:
                # 如果上一个更大，我们需要交换
                node.prev.value, node.value = node.value, node.prev.value
                # 这表示我们需要再次扫描
                is_sorted = False
            node = node.next

        # 它在顶部重置过，但是如果我们没有交换，那么它是有序的
        if is_sorted: break
```

我在这里添加了其他注释，以便你可以学习并跟踪它，将我在此处完成的内容与伪代码进行比较。你还应该看到，维基百科页面正在使用的数据结构，与 `DoubleLinkedList` 完全不同。维基百科的代码假设在数组或列表结构上实现函数。你必须将下面这行：

```
if A[i-1] > A[i] then
```

使用 `DoubleLinkedList` 翻译为 Python：

```python
if node.prev.value > node.value:
```

我们不能轻易地随机访问 `DoubleLinkedList`，所以我们必须将这些数组索引操作转换为 `.next` 和 `.prev`。在循环中，我们还必须注意 `next` 或 `prev` 属性是否是 `None`。这种转换需要大量的翻译，学习和猜测你正在阅读的伪代码的语义。

学习冒泡排序

你现在应该花时间研究这个 `bubble_sort` Python 代码，看看我如何翻译它。确保观看我实时的视频，并获得更多的透视。你还应该绘制在不同类型的列表（已排序，随机，重复等）上运行的图表。一旦你了解我是如何做到的，为此研究 `pytest` 和 `merge_sort` 算法：

```python
import sorting
from dllist import DoubleLinkedList
from random import randint

max_numbers = 30

def random_list(count):
    numbers = DoubleLinkedList()
    for i in range(count, 0, -1):
        numbers.shift(randint(0, 10000))
    return numbers


def is_sorted(numbers):
    node = numbers.begin
    while node and node.next:
        if node.value > node.next.value:
            return False
        else:
            node = node.next

    return True


def test_bubble_sort():
    numbers = random_list(max_numbers)

    sorting.bubble_sort(numbers)

    assert is_sorted(numbers)


def test_merge_sort():
    numbers = random_list(max_numbers)

    sorting.merge_sort(numbers)

    assert is_sorted(numbers)
```

这个测试代码的一个重要部分是，我正在使用 `random.randint` 函数生成随机数据进行测试。这个测试不会测试许多边界情况，但这是一个开始，我们将在以后进行改进。记住，你没有实现 `sort.merge_sort`，所以你可以不写这个测试函数，或者现在注释它。

一旦你进行了测试，并且写完了这个代码，再次研究维基百科页面，然后在尝试 `merge_sort` 之前，尝试一些其他的 `bubble_sort` 版本。

归并排序

我还没准备好让你自己实现它。我将再次对 `merge_sort` 函数重复此过程，但是这次我想让你尝试，从归并排序的维基百科页面 上的伪代码中实现该算法，然后再查看我怎么做。有几个建议的实现，但我使用“自顶向下”的版本：

```
function merge_sort(list m)
    if length of m ≤ 1 then
        return m

    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    left := merge_sort(left)
    right := merge_sort(right)

    return merge(left, right)

function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

为 `test_merge_sort` 编写剩余测试用例函数，然后在这个实现上进行尝试。我会给你一个线索，当仅仅使用第一个 `DoubleLinkedListNode` 时，该算法效果最好。你也可能需要一种方法，从给定的节点计算节点数。这是 `DoubleLinkedList` 不能做的事情。

归并排序作弊模式

如果你尝试了一段时间并且需要作弊，这是我所做的：

```python
def count(node):
    count = 0

    while node:
        node = node.next
        count += 1

    return count


def merge_sort(numbers):
    numbers.begin = merge_node(numbers.begin)

    # horrible way to get the end
    node = numbers.begin
    while node.next:
        node = node.next
    numbers.end = node


def merge_node(start):
    """Sorts a list of numbers using merge sort."""
    if start.next == None:
        return start

    mid = count(start) // 2

    # scan to the middle
    scanner = start
    for i in range(0, mid-1):
        scanner = scanner.next

    # set mid node right after the scan point
    mid_node = scanner.next
    # break at the mid point
    scanner.next = None
    mid_node.prev = None

    merged_left = merge_node(start)
    merged_right = merge_node(mid_node)

    return merge(merged_left, merged_right)



def merge(left, right):
    """Performs the merge of two lists."""
    result = None

    if left == None: return right
    if right == None: return left

    if left.value > right.value:
        result = right
        result.next = merge(left, right.next)
    else:
        result = left
        result.next = merge(left.next, right)

    result.next.prev = result
    return result
```

在尝试实现它时，我将使用此代码作为“备忘单”来快速获取线索。你还会看到，我在视频中尝试从头开始重新实现此代码，因此你可以看到我努力解决你可能遇到过的相同问题。

快速排序

最后，轮到你尝试实现`quick_sort`并创建`test_quicksort`测试用例。我建议你首先使用 Python 的普通列表类型实现简单的快速排序。这将有助于你更好地理解它。然后，使用简单的 Python 代码，并使其处理`DoubleLinkedList`（的头节点）。记住要把你的时间花费在这里，显然还要在你的`test_quicksort`里进行大量的调试和测试。

深入学习

- 这些实现在性能上绝对不是最好的。尝试写一些丧心病狂的测试来证明这一点。你可能需要将一个很大的列表传给算法。使用你的研究来找出病态（绝对最差）的情况。例如，当你把一个有序的列表给`quick_sort`时会发生什么？
- 不要实现任何改进，但研究你可以对这些算法执行的，各种改进方法。
- 查找其他排序算法并尝试实现它们。
- 它们还可以在`SingleLinkedList`上工作吗？`Queue`和`Stack`呢？它们很实用吗？
- 了解这些算法的理论速度。你会看到`O(n^2)`或`O(nlogn)`的引用，这是一种说法，在最坏的情况下，这些算法的性能很差。为算法确定“大O”超出了本书的范围，但我们将在练习 18 中简要讨论这些度量。
- 我将这些实现为一个单独的模块，但是将它们作为函数，添加到`DoubleLinkedList`更简单吗？如果你这样做，那么你需要将该代码复制到可以处理的其他数据结构上吗？我们没有这样的设计方案，如何使这些排序算法处理任何“类似链表的数据结构”。
- 再也不要使用气泡排序。我把它包含在这里，因为你经常遇到坏的代码，并且我们会在练习 19 中提高其性能。

练习 17：字典

> 原文：Exercise 17: Dictionary
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你应该熟悉 Python 的 `dict` 类。无论什么时候，你编写这样的代码：

```
cars = {'Toyota': 4, 'BMW': 20, 'Audi': 10}
```

你在使用字典，将车的品牌（“丰田”，“宝马”，“奥迪”）和你有的数量（4，20，10）关联起来。现在使用这种数据结构应该是你的第二本能，你可能甚至不考虑它是如何工作的。在本练习中，你将通过从已经创建的数据结构，实现自己的 `Dictionary` 来了解 `dict` 的工作原理。你在本练习中的目标是，根据我在这里写的代码实现自己的 `Dictionary` 版本。

挑战性练习

在本练习中，你将完全记录并理解我编写的一段代码，然后尽可能地，根据记忆编写自己的版本。本练习的目的是，学习剖析和理解复杂的代码。能够内在化或记忆，如何创建一个简单的数据结构（如字典）是很重要的性。我发现，学习剖析和理解一段代码的最好方法是，根据自己的学习和记忆来重新实现它。

将其看做一个“原件”类。原件来自绘画，其中你绘制一幅由他人创作的画，优于创作它的副本。这样做会教你如何绘画并且提高你的技能。代码和绘画是相似的，因为所有的信息都为复制准备好了，所以你可以通过复制他们的工作，轻松地向别人学习。

制作一份“代码大师的副本”

要创建一份“代码大师副本”，你将遵循这个的流程，我称之为 CASMIR 流程：

- 复制代码，使其正常工作。你的副本应该完全一样。这有助于你了解它，并强制你仔细研究它。
- 使用注释来标注代码，并为所有代码写一个分析，确保你了解每一行以及它的作用。这可能涉及到你编写的其他代码，来将整个概念结合在一起。
- 使用简洁的说明，为这个代码的工作原理总结一般结构。这是函数列表和每个函数的作用。
- 记住这个算法和关键代码段的简洁描述。

- 根据记忆实现可以实现的东西，当你用尽细节时，回顾你的笔记和原始代码来记住更多内容。
- 当你需要从你的记忆中复制的时候，重复此过程多次。你的记忆中的副本并不必须是完全一样的，但应接近，并通过你创建的相同测试。

这样做将使你深入了解数据结构的工作原理，但更为重要的是，帮助你内在化和回忆此数据结构。你终将能够理解该概念，并在需要创建数据结构时实现数据结构。这也是训练你的大脑，在未来记住其他的数据结构和算法。

> 警告
>
> 我要做的唯一的警告是，这是一个很简单，愚蠢，缓慢的 `Dictionary` 实现。你真的复制了一个简单愚蠢的 `Dictionary`，它具有所有的基本元素和作用，但需要大量改进来用于生产。当我们到达练习 19 并研究性能调整时，会进行这些改进。现在，只需实现这个简单的版本，就可以了解数据结构的基础知识。

复制代码

首先我们查看 `Dictionary` 的代码，你需要复制它：

```python
from dllist import DoubleLinkedList

class Dictionary(object):
    def __init__(self, num_buckets=256):
        """Initializes a Map with the given number of buckets."""
        self.map = DoubleLinkedList()
        for i in range(0, num_buckets):
            self.map.push(DoubleLinkedList())

    def hash_key(self, key):
        """Given a key this will create a number and then convert it to
        an index for the aMap's buckets."""
        return hash(key) % self.map.count()

    def get_bucket(self, key):
        """Given a key, find the bucket where it would go."""
        bucket_id = self.hash_key(key)
        return self.map.get(bucket_id)

    def get_slot(self, key, default=None):
        """
        Returns either the bucket and node for a slot, or None, None
        """
        bucket = self.get_bucket(key)

        if bucket:
            node = bucket.begin
            i = 0

            while node:
                if key == node.value[0]:
                    return bucket, node
                else:
                    node = node.next
                    i += 1

        # fall through for both if and while above
        return bucket, None
```

```
    def get(self, key, default=None):
        """Gets the value in a bucket for the given key, or the default."""
        bucket, node = self.get_slot(key, default=default)
        return node and node.value[1] or node

    def set(self, key, value):
        """Sets the key to the value, replacing any existing value."""
        bucket, slot = self.get_slot(key)

        if slot:
            # the key exists, replace it
            slot.value = (key, value)
        else:
            # the key does not, append to create it
            bucket.push((key, value))

    def delete(self, key):
        """Deletes the given key from the Map."""
        bucket = self.get_bucket(key)
        node = bucket.begin

        while node:
            k, v = node.value
            if key == k:
                bucket.detach_node(node)
                break

    def list(self):
        """Prints out what's in the Map."""
        bucket_node = self.map.begin
        while bucket_node:
            slot_node = bucket_node.value.begin
            while slot_node:
                print(slot_node.value)
                slot_node = slot_node.next
            bucket_node = bucket_node.next
```

该代码使用你现有的 `DoubleLinkedList` 代码来实现 `dict` 数据结构。如果你不完全了解 `DoubleLinkedList`，那么你应该尝试使用代码复制过程，让我们更好地理解它。一旦你确定你了解 `DoubleLinkedList`，你可以键入此代码并使其正常工作。记住，在开始标注之前，它必须是完美的副本。你可以做的最糟糕的事情，是标注我的代码的破损或不正确的副本。

为了帮助你获得正确的代码，我写了一个快速和简陋的小型测试脚本：

```python
from dictionary import Dictionary

# create a mapping of state to abbreviation
states = Dictionary()
states.set('Oregon', 'OR')
states.set('Florida', 'FL')
states.set('California', 'CA')
states.set('New York', 'NY')
states.set('Michigan', 'MI')

# create a basic set of states and some cities in them
cities = Dictionary()
cities.set('CA', 'San Francisco')
cities.set('MI', 'Detroit')
cities.set('FL', 'Jacksonville')

# add some more cities
cities.set('NY', 'New York')
cities.set('OR', 'Portland')


# print(out some cities
print('-' * 10)
print("NY State has: %s" % cities.get('NY'))
print("OR State has: %s" % cities.get('OR'))

# print(some states
print('-' * 10)
print("Michigan's abbreviation is: %s" % states.get('Michigan'))
print("Florida's abbreviation is: %s" % states.get('Florida'))

# do it by using the state then cities dict
print('-' * 10)
print("Michigan has: %s" % cities.get(states.get('Michigan')))
print("Florida has: %s" % cities.get(states.get('Florida')))

# print(every state abbreviation
print('-' * 10)
states.list()

# print(every city in state
print('-' * 10)
cities.list()

print('-' * 10)
state = states.get('Texas')

if not state:
  print("Sorry, no Texas.")

# default values using ||= with the nil result
# can you do this on one line?
city = cities.get('TX', 'Does Not Exist')
print("The city for the state 'TX' is: %s" % city)
```

我希望你也可以正确地键入这个代码，但是当你进入大师副本的下一个阶段时，你会把它变成一个正式的自动测试，你可以运行 `pytest` 。现在，只要让这个脚本工作，就可以让 `Dictionary` 类工作，之后你可以在下一个阶段清理它。

标注代码

确保我的代码的副本完全一样，并通过测试脚本。然后，你可以开始标注代码，并研究每一行来了解其作用。一个非常好的方式是，编写一个“正式”的自动化测试，并在你工作时标注代码。获取`dictionary_test.py`脚本，并将每个部分转换成一个小型测试函数，然后标注`Dictionary`类。

例如，`test_dictionary.py`中的第一部分测试创建一个字典，并执行一系列`Dictionary.set`调用。我会把它转换成一个`test_set`函数，然后在`dictionary.py`文件中标注`Dictionary.set`函数。当你标注`Dictionary.set`函数时，你必须潜入到`Dictionary.get_slot`函数中，然后是`Dictionary.get_bucket`函数，最后是`Dictionary.hash_key`。这迫使你通过一个测试和有组织的方式，来标注和了解`Dictionary`类的大段代码。

总结数据结构

你现在可以总结你在`dictionary.py`中，通过标注代码所学到的内容，并将`dictionary_test.py`文件重写为真正的`pytest`自动测试。你的摘要应该是数据结构的清晰和细微描述。如果你可以把它写在一张纸上，那么你做得很好。并不是所有的数据结构都可以简明扼要地总结出来，但是保持摘要简洁将有助于你记住它。你可以使用图表，图纸，单词，或你能够记住的任何内容。

此摘要的目的是为你提供一组快速注解，你可以“挂载”更多的细节，当你的记忆进行到下一步的时候。摘要不一定包括所有内容，但应该包括一些细节，可以触发你对“标注”阶段的代码的记忆，从而触发你对“复制”阶段的记忆。这被称为“分块”，你可以将更详细的记忆和信息附加到信息的细微碎片。在撰写摘要时记住这一点。少即是多，但太少没有用。

记忆摘要

你可以用任何方式记住摘要和带标注的代码，但我将给出一个基本的记忆过程，你可以使用它。老实说，记住复杂的东西是每个人的不断尝试和犯错的过程，但有些技巧有帮助：

- 确保你有一个纸质的笔记本，以及摘要和代码的打印。
- 花3分钟，只需阅读摘要并尝试记住它。静静地看着它，大声读出来，然后闭上眼睛，重复你所读的内容，甚至尝试记住纸上的单词的“形状”。听起来很愚蠢，但相信我，它完全奏效。记住你的大脑比你想象的更好。
- 把摘要翻过来，并尝试从你记住的内容中再次写出来，当你卡住时，将其快速翻过来并查看。在你快速瞥见之后，把摘要翻过来，并尝试完成更多。
- 一旦从（大部分）记忆中写出了摘要的副本，请使用摘要，花另一个 3 分钟，试图记住带标注的代码。仅仅阅读摘要的一部分，然后再看看代码的相关部分，并尝试记住它。甚至每个函数只能花 3 分钟。
- 一旦你花时间试图记住带标注的代码，把它翻过去，使用摘要，尝试回忆你笔记本中的代码。同样，当你陷入困境时，快速把标注翻过来并查看。

- 继续这样做，直到你可以在纸上写出代码的完整副本。你纸上的代码不一定是完美的 Python 代码，但应该非常接近原始代码。

看起来这可能是无法实现，但是当你这么做时，你会感到惊讶。完成此操作后，你也会惊讶于你了解了字典的概念。这不是简单的记忆，而是建立一个概念图，当你尝试自己实现字典时，你可以实际使用它。

> 警告
>
> 如果你是那种担心记不住任何东西的人，那么这个练习会为你将来带来巨大的帮助。能够遵循流程来记住某些东西，有助于克服任何记忆的挫折。你并不是沉浸在“失败”中，而是可以在坚持中看到缓慢的改进。当你这样做，你会看到改善你的回忆的方式和黑魔法，并且你会做得更好。你只需要相信我，这似乎是一种缓慢的学习方式，但它比其他技术要快得多。

从记忆中实现

现在是时候走到你的电脑旁边 - 把你的纸质笔记放在另一个房间或地板上 - 并根据记忆尝试你的第一个实现。你的第一次尝试可能完全是一场灾难，也可能完正确。你最可能不习惯从记忆中实现任何东西。只要放下任何你记得的东西，当你到达你的记忆的彼端，回到另一个房间，记忆更多东西。经过几次到你的记忆空间的旅行，你会进入它，记忆会更好地流出来。你完全可以再次访问你的记忆笔记。这一切都关于，试图保持代码的记忆并提高自己的技能。

我建议你首先写下你的想法，无论是测试，代码还是两者。然后使用你可以回忆的内容，来实现或回忆代码的其他部分。如果你首先坐下来并记住`test_set`函数名和几行代码，然后把它们写下来。当他们在你的头脑中，立即利用它们。一旦你完成了，尽你最大的努力，使用这个测试来记住或实现`Dictionary.set`函数。你的目标是使用任何信息来构建或者其它信息。

你也应该尝试，用你对`Dictionary`的理解来实现代码。不要简单以摄影方式来回忆每一行。这实际上是不可能的，因为没有人有摄影记忆（去查一下，没有人）。大多数人的记忆都不错，能够触发他们可以使用的概念性理解。你应该做同样的事情，并使用你的`Dictionary`的知识来创建自己的副本。在上面的示例中，你知道`Dictionary.set`以某种方式运行，你需要一种方法来获取插槽（链表节点）和桶（链表本身）...所以这意味着，你需要`get_slot`和`get_bucket`。你不是以摄影方式记住每个字符；而是记住所有关键概念并使用它们。

重复

这个练习最重要的部分是，重复几次这个流程，使其没有错误，才能使其更好。你会对这本书中的其他数据结构这样做，所以你会得到大量的练习。如果你必须回去记忆 100 次才行，也是可以的。最终你只需要做 50 遍，然后下一次只有 10 遍，然后最终你将能够轻易从记忆中实现一个 `Dictionary` 。尽管继续尝试，并尝试像冥想那样接近它，所以你这样做的时候可以放松。

深入学习

- 我的测试非常有限。写一个更广泛的测试。
- 练习 16 的排序算法如何有助于这个数据结构？
- 当你将键和值随机化，用于这个数据结构时，会发生什么？排序算法有帮助吗？
- `num_buckets` 对数据结构有什么影响？

破坏它

你的大脑可能宕机了，要休息一下，然后尝试破坏这个代码。这个实现很容易被数据淹没和压倒。奇怪的边界情况如何呢？你可以将任何东西添加为一个键，或者只是字符串？会造成什么问题？最后，你是否可以对代码暗中耍一些花招，使其看起来像是正常工作，但实际上是以一些机智的方式来破坏它？

练习 18：性能测量

> 原文：Exercise 18: Measuring Performance
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

在本练习中，你将学习使用多种工具来分析你创建的数据结构和算法的性能。为了使这个介绍专注并且简洁，我们将查看练习 16 中的 `sorted.py` 算法的性能，然后在视频中，我会分析我们迄今为止所做的所有数据结构的性能。

性能分析和调优是我最喜欢的计算机编程活动之一。在看电视的时候，我是那个手里拿着一团缠着的绳子的人，并且只打算把它解开，直到它很好并且有序。我喜欢探究复杂的奥秘，代码性能是最复杂的奥秘之一。有一些很好的并且实用的工具，用于分析代码的性能，使之比调试更好。

编码时不要试图实现性能改进，除非它们是显而易见的。我更喜欢使我的代码的初始版本保持极其简单和朴素，以便我可以确保它正常工作。然后，一旦它运行良好，但也许很慢，我启动我的分析工具，并开始寻找方法使其更快，而不降低稳定性。最后一部分是关键，因为许多程序员觉得如果能使代码更快，那么可以降低代码的稳定性和安全性。

工具

在本练习中，我们将介绍许多有用的 Python 工具，以及一些改进任何代码性能的一般策略。我们将使用的工具有：

- `timeit`
- `cProfile` 和 `profile`

在继续之前，请确保安装任何需要安装的软件。然后获取 `sorted.py` 和 `test_sorting.py` 文件的副本，以便我们可以将这些工具应用到这些算法中。

`timeit`

`timeit` 模块不是非常有用。它所做的就是接受字符串形式的 Python 代码，并使用一些时间运行它。你不能传递函数引用，`.py` 文件或除字符串之外的任何内容。我们可以在 `test_sorting.py` 的结尾，测试 `test_bubble_sort` 函数需要多长时间：

```
if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test_bubble_sort()", setup="from __main__ import test_bubble_
sort"))
```

它也不会产生有用的测量或任何信息，为什么某些东西可能很慢。我们需要一种方式来衡量代码运行的时间长短，这样做太笨重了，无法使用。

`cProfile` 和 `profile`

接下来的两个工具，对于测量代码的性能来说更为有用。我建议使用 `cProfile` 来分析代码的运行时间，并且当你在分析中需要更多的灵活性时，保存 `profile`。为了对你的测试运行 `cProfile`，请更改 `test_sorting.py` 文件的末尾，来简单地运行测试函数：

```
if __name__ == '__main__':
    test_bubble_sort()
    test_merge_sort()
```

并将 `max_numbers` 更改为大约 800，或足够大的数字，以便你可以测量效果。一旦你完成了，然后在你的代码上运行 `cProfile`：

```
$ python -m cProfile -s cumtime test_sorting.py | grep sorting.py
```

我使用了 `| grep sorted.py`，只是将输出缩小到我关心的文件，但删除该部分命令可以查看完整的输出。我在相当快的电脑上获得的 800 个数字的结果是：

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.145    0.145 test_sorting.py:1(<module>)
        1    0.000    0.000    0.128    0.128 test_sorting.py:25(test_bubble_sort)
        1    0.125    0.125    0.125    0.125 sorting.py:6(bubble_sort)
        1    0.000    0.000    0.009    0.009 sorting.py:1(<module>)
        1    0.000    0.000    0.008    0.008 test_sorting.py:33(test_merge_sort)
        2    0.001    0.000    0.006    0.003 test_sorting.py:7(random_list)
        1    0.000    0.000    0.005    0.005 sorting.py:37(merge_sort)
   1599/1    0.001    0.000    0.005    0.005 sorting.py:47(merge_node)
 7500/799    0.004    0.000    0.004    0.000 sorting.py:72(merge)
      799    0.001    0.000    0.001    0.000 sorting.py:27(count)
        2    0.000    0.000    0.000    0.000 test_sorting.py:14(is_sorted)
```

我在顶部添加了标题，以便你看到输出表示什么。每个标题的意思是：

> `ncalls`
>
> 该函数的调用次数
>
> `tottime`
>
> 总执行时间
>
> `percall`
>
> 函数每个调用的总时间
>
> `cumtime`
>
> 函数的累计时间
>
> `percall`
>
> 每个调用的累计时间
>
> `filename:lineno(function)`
>
> 名称、行号和涉及到的函数

那些标题名称也可以使用 `-s` 参数来获取。然后，我们可以对此输出进行快速分析：

`bubble_sort` 被调用一次，但 `merge_node` 被调用了 1599 次，并且 `merge` 甚至调用了 7500 次。这是因为 `merge_node` 和 `merge` 是递归的，所以对一个有 800 个元素的随机列表排序时，他们会产生大量的调用。

即使 `bubble_sort` 不像 `merge` 或 `merge_node` 一样被频繁调用，它也是很慢的。这符合两种算法的性能预期。归并排序的最坏情况是 `O(nlogn)` ，但是对于冒泡排序，它是 `O(n^2)` 。如果你有 800 个元素，那么 `800 * log(800)` 约为 5347，而 `800^2` 是 640000！这些数字不一定会转化为这些算法运行的精确秒数，但它们确实会转化为相对比较。

`count` 函数被调用 799 次，这最有可能是巨大的浪费。我们实现的 `DoubleLinkedList` 并不追踪元素的数量，而是必须在每一次你想知道数量的时候遍历这个列表。我们在这里的 `count` 函数中使用相同的方法，并且导致了整个列表中的 800 个元素的 799 次遍历。将 `max_numbers` 更改为 600 或 500 在这里查看规律。注意在我们的实现中， `count` 是否运行了 `n-1` 次？这意味着我们遍历了几乎所有 800 个元素。

现在让我们查看， `dllist.py` 如何影响其性能：

同样，我已经添加了标题，以便你可以看到发生了什么。在这种情况下，你可以看到，与 `merge` ， `merge_node` 和 `count` 函数相比， `dllist.py` 函数不会影响性能。这是很重要的，因为大多数程序员将运行优化 `DoubleLinkedList` 数据结构，但在 `merge_sort` 实现中可以获得更大的收益，并且完全可以避免使用 `bubble_sort` 。始终以最小的努力获得最大的改进。

性能分析

分析性能只是一件事情，找出什么较慢，然后试图确定为什么它较慢。它类似于调试，除了你最好不要改变代码的行为。完成后，代码的工作方式应该完全一样，仅仅是更快执行。有时修复性能也会发现错误，但是当你尝试加速时，最好不要尝试完全重新设计。一次只做一件事。

在开始分析性能之前，另一件重要的事情是，软件所需的一些指标。通常快即是好，但没有目标，你最终会提出一些完全不必要的解决方案。如果你的系统以 50 个请求/秒执行，并且你真的只需要 100 个请求/秒，那么没有必要使用 Haskell 完全重写它，来获得 200 的性能。这个过程完全关于，“节省最多的钱，并且付出最少的努力”，并且你需要某种测量作为目标。

你可以从运营人员那里获得大部分测量结果，并且应该有很好的图表，显示了 CPU 使用情况，请求/秒，帧速率，任何他们或客户认为重要的东西。然后，你可以与他们一起设计测试，证明一些缓慢的东西需要定位，以便你可以改进代码来达到所需的目标。你可以从系统中榨取更多的性能，从而节省资金。你可以尝试并得出结论，这只是一个需要更多 CPU 资源的难题。有了一个作为目标的指标，你会明白什么时候放弃，或已经做得足够了。

你可以用于分析的最简单过程是这样：

- 在代码上运行性能分析器，就像我在这里使用测试所做的一样。你得到的信息越多越好。有关免费的其他工具，请参阅深入学习部分。向人们询问一些工具，它们用于分析系统的速度。
- 识别最慢和最小的代码段。不要编写一个巨大的函数，并尝试分析它。很多时候这些函数很慢，因为它们使用了一大堆其他很慢的函数。首先找到最慢和最小的函数，你最有可能得到最大的收益，并付出最少的努力。
- 审查这些缓慢的代码，和任何他们接触的代码，寻找代码缓慢的可能原因。循环内有循环吗？调用函数太频繁吗？在调查诸如缓存之类的复杂技术之前，寻找可以改变的简单事物。
- 一旦你列出了所有最慢和最小的函数，以及简单的更改，使它们更快并寻找规律。你能在其它你看不到的地方做这件事吗？
- 最后，如果没有简单更改你可以更改的小函数，可以寻求可能的较大改进。也许真的是完全重写的时候了吗？不要这样做，直到你至少尝试了简单的修复。
- 列出你尝试的所有东西，以及你所完成的所有性能增益。如果你不这样做，那么你会不断地回到你已经处理过的函数上，并浪费精力。

在这个过程中，“最慢和最小”的概念是变化的。你修复了十几个 10 行的函数并使其更快，这意味着现在你可以查看最慢的 100 行的函数。一旦你让 100 行的函数运行得更快，你可以查看正在运行的更大的一组函数，并提出使其加速的策略。

最后，加速的最好办法是完全不做。如果你正在对相同条件进行多重检查，请找到避免多次检查的方法。如果你反复计算数据库中的同一列，请执行一次。如果你在密集的循环中调用函数，但数据不怎么改变，请缓存它或者事先计算出来。在许多情况下，你可以通过简单地事先计算一些东西，并一次性存储它们，来用空间换时间。

在下一个练习中，我们将会使用这个过程，来改进这些算法的性能。

挑战练习

此练习的挑战是，将我对`bubble_sort`和`merge_sort`所做的所有操作，都应用到目前为止所创建的所有数据结构和算法。我不期望你改进他们，但只是在开发测试来显示性能问题时，记下笔记并分析性能。抵制现在修改任何东西的诱惑，因为我们将在练习 19 中提高性能。

研究性学习

- 到目前为止，对所有代码运行这些分析工具，并分析性能。
- 将结果与算法和数据结构的理论结果进行比较。

破坏它

尝试编写使数据结构崩溃的病态测试。你可能需要为他们提供大量数据，但使用性能分析的信息来确保正确。

深入学习

- 查看`line_profiler`，它是另一个性能测量工具。它的优点是，你只能衡量你关心的函数，但缺点是你必须更改源代码。
- `pyprof2calltree`和`KCacheGrind`是更先进的工具，但老实说只能在 Linux 上工作。在视频中，我演示在 Linux 下使用它们。

练习 19：改善性能

> 原文：Exercise 19: Improving Performance
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

这几乎完全是视频练习，其中我演示了如何改进你至今为止编写的代码的性能，但首先你应该尝试它。你已经分析了 练习 18 的代码的速度有多慢，所以现在是时候实现你的一些想法。修复简单的性能问题时，我会给你一个简单的列表来寻找和修改：

- 循环内的循环的重复计算可以避免。冒泡排序是经典案例，这就是我教它的原因。，一旦你看到，冒泡排序与其他方法相比有多糟糕，你将开始认识到这是一个需要避免的常见模式。
- 重复计算一些没有实际变化的东西，或者在更改过程中可以计算一次。在 `sorted.py` 和其他数据结构中的 `count()` 函数是一个很好的例子。你可以在函数内跟踪数据结构的大小。每次添加时，你可以增加它，并且每次删除时，减少它。每次都不需要遍历整个列表。你还可以使用这个预先计算的计数，通过检查 `count == 0` 来改进其他功能的逻辑。
- 使用错误的数据结构。在字典中，我使用 `DoubleLinkedList` 来演示这个问题。字典需要随机访问元素，至少是桶的列表中的元素。使用 `DoubleLinkedList` 的 `DoubleLinkedList` 意味着每次你想访问第 n 个元素，你必须遍历所有元素直到 n。用 Python 列表替换它将大大提高性能。这是一个练习，使用现有代码从更简单的数据结构中构建数据结构，因此不一定是实现最好的 Python `Dictionary`（它已经有一个了）的练习。
- 对数据结构使用错误的算法。冒泡排序显然是错误的算法（不要再使用了），但要记住归并排序和快速排序是否更好，这可能取决于数据结构。归并排序对于这些类型的链接数据结构来说是非常好的，但对于 Python `list` 之类的数组却不是很好。快速排序对于 `list` 更好，但在链接的数据结构上不是很好。
- 不在最佳的地方优化常见的操作。在 `DoubleLinkedList` 中，你将经常从桶的开头开始，并在槽中搜索一个值。在当前的代码中，这些槽进来时，你简单地添加它们，这可能是随机的也可能不是。如果你采取了一个规则，在插入时排序这些列表，那么寻找元素会更容易和更快捷。当槽的值大于你要查找的值时，你可以停止，因为你知道它是有序的。这样做使得插入速度更慢，但使几乎每一个其它操作变快，因此要为练习选择正确的设计。如果你需要执行大量的插入，那么这不是很机智。但是，如果你的分析显示，你需要执行很少的插入，但是很多的访问，这是个加速的不错方式。
- 手写代码，而不是使用现有的代码。我们正在做练习来学习数据结构，但在现实世界中，你不会这样做。Python 已经有很好的数据结构，内置在语言中并进行了优化。你应该首先使用这些，如果性能分析表明你自己的数据结构会更快，那么编写自己的数据结

构。即使这样，你应该查找一个现有的数据结构，其他人使其能工作，而不是手写自己的东西。在这个练习中，写一些测试，将你的 `Dictionary` 和 Python 内置类型 `list` 比较，看看你可能有多少优势。

- 在不太擅长的语言中使用递归。简单地说，`merge_sort` 代码可以通过给它一个比 Python 堆栈更大的列表，来使其崩溃。尝试给它一些丧心病狂的东西，例如 3000 个元素的列表，然后慢慢地减少元素数量，直到找到导致 Python 耗尽堆栈的极限值。Python 不执行某些递归优化，所以没有特别考虑的递归会像这样失败。在这种情况下，重写 `merge_sort` 来使用循环会更好（但要困难得多）。

在练习 18 的分析过程中，你应该有了一些很大的收获。现在你的任务是尝试实现它们，以及提升代码的性能。

挑战练习

尝试使用你的分析和上述建议性改进的描述，来系统地提升代码的性能。“系统地”的含义是，使用锁定步骤控制的方法来完成，使用数据来确认你已经改进了一些东西。这是你在此练习中遵循的流程：

- 选择你的第一个，最小、最慢的代码，并确保有一个测试来告诉你它有多慢。确保你有一系列的度量，让你了解其速度。如果可以的话，绘制出来。
- 尝试提升速度，然后重新运行测试。继续尝试压榨这段代码的所有的性能。
- 如果你尝试更改代码，并且不会改进任何事情，那么你可以确定你做错了，并且撤销该更改并尝试其他操作。这很重要，因为你正在验证假设，所以如果你在其中留下无用的代码更改，可能会改变你可以修复的，其他函数的性能。撤销更改并尝试不同的方法，或转向另一段代码。
- 重新测量其他最小最慢的代码片段，看看它们是否已更改。你的修复可能已修复了其他代码，因此重新确认你认为自己知道的东西。
- 一旦你完成了你确认的一切，再次运行你的测量，并选择新的代码段来尝试改进。
- 从第 1 步开始保持测试（他们应该是自动测试），因为你需要避免退步。如果你看到一个函数的修改，导致其他函数变慢，那么要么修复它，要么简单地撤销修改，并尝试一些新的方法。

深入学习

你应该研究 Tim Sort 的原始邮件，最后研究由 EU FP7 ENVISAGE 研究人员在 2015 年发现的错误。原始电子邮件于 2002 年发送，随后实现。这个 bug 发现了 13 年了。当你去实现自己的算法想法时，记住这一点。即使大型项目的顶尖开发人员也会在它们的算法中遗留 bug，它们很长时间都没有发现。另一个例子是 OpenSSL 项目，它几十年来一直存在 bug，因为每个人都相信“专业密码学家”创建了代码。原来，即使是所谓的专业密码学家也可以写出糟糕的代码。使新的算法正确需要特殊技能，并且我认为 -- 使用定理证明工具来验证正确

性。除非你有这样的背景，创造新的算法和数据结构可能会产生危险。这包括加密算法和加密网络协议。只要你掌握实现技能，实现其他人已经证明的算法完全正常，运行良好。但是不要在没有一些帮助的情况下制作自己的头发数据结构。实现其他人已经证明的算法完全没问题，并且是个好的练习。但是不要在没有一些帮助的情况下制作自己的粗制滥造的数据结构。

练习 20：二叉搜索树

> 原文：Exercise 20: Binary Search Trees
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在本练习中，我将让你将数据结构的中文描述翻译成工作代码。你已经知道如何使用“大师复制”方法，分析算法或数据结构的代码。你还可以了解如何阅读算法的伪代码描述。现在你将结合二者，并学习如何拆分一个相当松散的二进制搜索树的英文描述。

我打算马上开始，并提醒你，当你做这个练习的时候，不要访问维基百科页面。维基百科的二进制搜索树描述拥有可以工作的 Python 代码，因此它会使此练习失败。如果你卡住了，那么你可以阅读任何你可以使用的资源，但是首先尝试按照这里我的描述来实现。

二叉搜索树

在练习 16 中，你了解了“归并排序”接受扁平的链表，将其转换为已排序部分的树。它将列表切成小块，然后通过排序左侧较小值的部分，以及右侧较大值的部分，将其重新组合在一起。在某种程度上，二叉搜索树（`BSTree`）是一种数据结构，本身就是有序的，并且不会使用列表来储存元素。`BSTree` 的一个主要用途是，用一棵树来组织 `key = value` 节点的偶对，在你插入或者删除它们的时候，保持它们有序。

最开始，`BSTree` 拥有一个 `key=value` 根节点，它拥有左子节点或者右子节点（都是链接）。如果插入一个新的 `key=value`，那么 `BSTree` 的任务是，从根节点开始，将 `key` 与每一个节点进行比较：如果新的键小于或等于它，走左边；如果新的键大于它，走右边。最终，`BSTree` 在树中找到一个位置，如果你遵循原始路径，你应该按照相同的过程找到它。之后的所有操作都是一样的，通过将任何键与每个节点，左移或者右移，直到找到节点或到达末尾。

这样，`BSTree` 是练习 17 中的 `Dictionary` 的替代品，因此它应该具有相同的操作。基本的 `BSTreeNode` 将需要 `left`，`right`，`key` 和 `value` 属性来创建树结构。你可能还需要 `parent` 属性，具体取决于你如何执行此操作。（译者注：如果你在遍历过程中记录父节点，就不用这个属性。）然后，`BSTree` 需要在根 `BSTreeNode` 上进行以下操作：

> `get`
>
> 提供一个键，遍历树，找到节点，或者如果到达末尾，返回`None`。如果提供的键是小于等于节点的键，走左边。如果键大于节点的键，走右边。如果你碰到一个没有左子节点或右子节点的节点，那么你已经遍历完了，并且该节点不存在。可以使用递归或使用`while`循环。
>
> `set`
>
> 这和`get`几乎一样，除了一旦你到达末尾的节点，你只需将一个新的`BSTreeNode`挂载到左子节点或右子节点，从而将树向下延伸了一个分支。
>
> `delete`
>
> 从`BSTree`删除节点是一个复杂的操作，所以我有一个完整的部分只是讲删除。简而言之有三个情况：节点是叶子（没有子节点），有一个子节点，或者有两个子节点。如果它是叶子，那么只是删除它。如果有一个子节点，然后将其替换为子节点。如果它有两个子节点，那么它变得非常复杂，因此请阅读下面删除的部分。
>
> `list`
>
> 遍历树，打印一切东西。`list`的重要内容是，你可以以不同的方式遍历树，Kauai产生不同的输出。如果你遍历`left`，之后是`right`，那么你会得到一些不同于反着执行的东西。如果你走了所有到底部的路，然后当你朝着`root`向上走的时候，打印结果，你会得到另一种类型的输出。你也可以在向下遍历树的时候打印节点，从`root`到“叶子”。尝试不同的风格，看看它们都做了什么。

删除

记住，删除节点时我们需要处理三个情况（我称之为`D`）：

- `D`节点是“叶子”节点，因为它有没有子节点（左子节点或者右子节点）。只需从父节点删除它。
- `D`节点只有一个子节点（左子节点或者右子节点，但不是二者）。在这种情况下，你可以将该子节点的值移动到`D`节点，然后删除该子节点。这有效地替换了`D`节点与子节点（或“将子节点向上移动”）。
- `D`节点有左子节点和右子节点，这意味着这时候需要做一些大的操作。首先，找到的`D.right`节点的最小子节点，成为`successor`。将`D.key`赋为`successor.key`，然后对`successor`的子节点使用它的键，做相同的删除操作。

你最有可能还需要`find_minimum`和`replace_node_in_parent`操作，来执行这两个操作。我提到你可能需要`parent`属性，具体取决于你实现它的方式。我会假设使用`parent`节点，因为这在大多数情况下更容易。

> 注意
>
> 每个人都讨厌树的删除操作。这是一个复杂的操作，甚至是我最喜欢的参考书，Steven S. Skiena 的《算法设计手册》都跳过了它，因为实现“看起来有点可怕”。如果你很难弄清楚 `delete`，不要气馁。

挑战练习

你将使用这个故意模糊的描述实现你的 `BSTree`。当你第一次尝试时，尝试不要看太多的参考，然后当你卡住时，去阅读他人的实现方式。这个练习的重点是，尝试从一个糟糕的描述中解决一个复杂的问题。

解决这个问题的窍门是，首先将英文段落翻译成粗糙的伪代码。然后将粗糙的伪代码转换为更精确的伪代码。一旦你有了更精确的伪代码，你可以把它翻译成 Python 代码。特别注意具体的单词，例如单个英文单词可能意味着 Python 中的很多东西。有时你只需要猜测并运行你的测试，看看是否正确。

测试也非常重要，对这个问题应用“测试第一”的方法，可能是一个好主意。你知道这些操作应该做什么，所以你可以为它编写一个测试，然后让测试工作。

研究性学习

- 你是否可以开发一个病态的测试，以某种方式插入元素，使 `BSTree` 只不过是一个花式链表？
- 当你尝试删除这个 `BSTree` 的“极点”时，会发生什么？
- 与你的最近优化的 `Dictionary` 相比，`BSTree` 的速度如何？
- 使用你的性能分析和调整流程，你能多快实现 `BSTree`？

练习 21：二分搜索

> 原文：Exercise 21: Binary Search
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

二分搜索算法是一个简单方法，在已排序的元素列表中查找元素。它很容易描述为接受排序列表，并将其分成两半，直到找到它或遍历完。如果你完成了练习 20，那么这个练习应该比较容易。

如果我们想在已排序的数值列表中找到数字 `X`，我们将这样做：

- 获取列表中间的数字（`M`）并将其与 `X` 进行比较。
- 如果 `X == M`，你就完成了。
- 如果 `X > M`，则在 `M + 1` 到列表末尾的区间内寻找。
- 如果 `X < M`，则在列表开头到 `M - 1` 的区间内寻找。
- 重复它，直到找到 `X` 或者区间为空。

这适用于任何可以比较相等性的东西。它适用于字符串，数字和任何你可以排序的东西。

挑战练习

你的BSTree应该已经有了一个 `get` 操作，类似于二分搜索。不同的是 `BSTree` 已经分块了，所以没有必要再这么做了。在本练习中，你将为 `DoubleLinkedList` 和Python `list` 实现二分搜索，并将其与 `BSTree.get` 的性能进行比较。你的目标是学习以下内容：

- 对于简单的寻找元素，`BSTree` 与 Python 的 `list` 相遇效果如何？
- `DoubleLinkedList` 的二分搜索有多糟糕？
- `BSTree` 的病态情况是否也会对 `list` 的二分搜索造成问题？

分析性能时，请不要包含排序数字所需的时间。这在进行全局优化时很重要，但在这种情况下，你只需要关心二分搜索的工作速度。你也可以使用 Python 内置列表的排序算法对 `list` 进行排序，因为这不是重点。这个练习完全关于，三种数据结构之间的搜索速度有多快。

研究性学习

- 找出该算法需要执行的，最大的可能的比较数量。首先尝试自己弄清楚，然后研究算法

来找出眞正的答案。之后记住眞正的答案。

- 这里的任何优化可以应用于排序算法吗？
- 尝试在每个数据结构中，可视化该算法正在做什么。例如，在 `DoubleLinkedList` 中，你几乎可以将其视为来回遍历，直到找到结果。
- 为了给自己一个额外的挑战，尝试使`DoubleLinkedList`成为一个有序的链表，其中每次插入始终在排序后的位置。现在编写你的性能分析，包括添加元素和排序数字列表，来了解如何提高总体性能。

深入学习

研究其他搜索算法，特别是字符串。因为 Python 的字符串的实现方式，其中许多将很难在 Python 中实现，但是试一试吧。

练习 22：后缀数组

> 原文：[Exercise 22: Suffix Arrays]()
>
> 译者：[飞龙]()
>
> 协议：[CC BY-NC-SA 4.0]()
>
> 自豪地采用[谷歌翻译]()

我想告诉你一个关于后缀数组的故事。在一段时间里，我正在西雅图的一家公司面试，当时好奇的是如何最有效地创建一个用于可执行二进制文件的 `diff` 。我的研究给我带来了后缀数组和后缀树。后缀数组只是，将字符串的所有后缀排序，储存到有序列表中。后缀树是类似的，但是比列表更像 `BSTree` 。这些算法相当简单，一旦你进行了排序操作，它们就具有很快的性能。他们解决的问题是，找到两个字符串之间最长的公共子串（或者在这种情况下是字节列表）。

你可以在 Python 中轻易创建一个后缀数组：

```
>>> magic = "abracadabra"
>>> magic_sa = []
>>> for i in range(0, len(magic)):
...     magic_sa.append(magic[i:])
...
>>> magic_sa
['abracadabra', 'bracadabra', 'racadabra', 'acadabra',
 'cadabra', 'adabra', 'dabra', 'abra', 'bra', 'ra', 'a']
>>> magic_sa = sorted(magic_sa)
>>> magic_sa
['a', 'abra', 'abracadabra', 'acadabra', 'adabra', 'bra',
 'bracadabra', 'cadabra', 'dabra', 'ra', 'racadabra']
>>>
```

正如你所看到的，我只是按顺序取下字符串的后缀，然后对列表进行排序。但是，这对我有什么用呢？一旦我有了这个列表，那么我可以通过这个列表的二分搜索，来找到我想要的任何后缀。这个例子很简陋，但是在实际的代码中，你可以很快地做到它，你可以跟踪所有的原始索引，所以你可以引用后缀的原始位置。它与其他搜索算法相比非常快，对于 DNA 分析等事情非常有用。

回到西雅图的面试。我在这个寒冷的房间被 C++ 程序员面试，为了一份 Java 工作。你可以断定，这不是一个非常有趣的面试，我绝对不会认为我会得到这份工作。在多年的时间中，我没有写过任何 C++，而且这个工作是针对 Java 的，当时我是一个 Java 专家。下一个面试官来了，他问我："如何在字符串中寻找子串？"

太棒了！我在空闲时间里一直在研究这个问题。我当然知道！我跳起来走到白板，向那个家伙解释如何制作一个后缀树，它如何提高搜索性能，修改后的堆排序如何更快，后缀树的工作原理，为什么它比三叉搜索树更好，以及如何在 C 中实现。我想，如果我可以展示如何在

C 中写出来，那么这将证明，我不只是一个核心能力的 Java 码工。

那个家伙很震惊，就像我在采访室里打开一袋新鲜的榴莲一样。他看着董事会，并且有些结巴，“呃，我是在寻找一些有关 Boyer-Moore 搜索算法的东西吗？你知道吗？我愁眉苦脸地说：“是啊，就像 10 年前一样。” 他摇摇头，拿着他的东西，起身说：“好吧，我会让大家知道我的想法。”

几分钟后，下一个面试官来了。他抬头看着白板，笑了起来并嘲笑我，然后问我另一个 C++ 模板元编程问题，我无法回答。我没有得到这份工作。

挑战练习

在这个练习中，你将会使用我的 Python 小会话并创建自己的后缀数组搜索类。该类将使用一个字符串，将其拆成后缀列表，然后对其进行以下操作：

> `find_shortest`
>
> 找到以它开始的最短子串。在上面的例子中，如果我搜索`abra`，那么它应该返回`abra`，而不是`abracadabra`。
>
> `find_longest`
>
> 找到以它开始的最长子串。如果我搜索`abra`，你返回`abracadabra`。
>
> `find_all`
>
> 查找以它开始的所有子串。这意味着`abra`返回`abra`和`abracadabra`。

你将需要对此进行良好的自动测试，并进行一些性能测量。我们将在以后的练习中使用它们。完成之后，你需要进行研究性学习来完成这个练习。

研究性学习

- 一旦你的测试正常工作，使用你的`BSTree`重写它，进行后缀排序和搜索。你还可以使用每个`BSTreeNode`的`value`，来跟踪原始字符串中存在该子串的位置。然后，你可以保留原始字符串。
- `BStree`如何为不同搜索操作更改你的代码？是否使其更简单或更难？

深入学习

彻底研究后缀数组及其应用。它们非常有用，但不是被大多数程序员熟知。

练习 23：三叉搜索树

> 原文：Exercise 23: Ternary Search Trees
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

我们将研究的最后一个数据结构称为三叉搜索树（TSTree），它可以在一组字符串中快速查找字符串。它类似于 `BSTree`，但是它有三个子节点，而不是两个，每个子节点只是一个字符而不是整个字符串。在 `BSTree` 中，左子节点和右子节点是树的“小于”和“大于”的分支。在 `TSTree` 中，左子节点，中子节点和右子节点是“小于”，“等于”和“大于”的分支。这可以让你选取一个字符串，将其分解成字符，然后遍历 `TSTree`，每次一个字符，直到找到它或者你到达了末尾。

通过将你要搜索的一组键拆成单个字符的节点，`TSTree` 高效地使用空间换取时间。每一个这些节点将占用比 `BSTree` 更多的空间，但这允许你仅仅通过比较键中的字符来搜索键。使用 `BSTree`，你必须比较每个节点的键和被搜索键中的大多数字符。使用 `TSTree`，你只需要比较被搜索键的每个字母，当你到达末尾，就完成了。

`TSTree` 的另一件不错的事情是，它知道一个键何时不存在于集合中。想象一下，你的键的长度为 10 个字符，你需要在一组其他的键中找到它，但是如果键不存在，则需要快速停止。使用 `TSTree`，你可以在一到两个字符的地方停止，到达树的末尾，并且知道这个键不存在。你最多只能比较键中的 10 个字符来发现它，字符比较比 `BSTree` 少得多。

挑战练习

这个练习中，你打算完成另一个“代码大师复制”的一部分，之后独立完成 `TSTree`。你所需的代码是：

```python
class TSTreeNode(object):

    def __init__(self, key, value, low, eq, high):
        self.key = key
        self.low = low
        self.eq = eq
        self.high = high
        self.value = value


class TSTree(object):

    def __init__(self):
        self.root = None

    def _get(self, node, keys):
        key = keys[0]
        if key < node.key:
            return self._get(node.low, keys)
        elif key == node.key:
            if len(keys) > 1:
                return self._get(node.eq, keys[1:])
            else:
                return node.value
        else:
            return self._get(node.high, keys)

    def get(self, key):
        keys = [x for x in key]
        return self._get(self.root, keys)

    def _set(self, node, keys, value):
        next_key = keys[0]

        if not node:
            # what happens if you add the value here?
            node = TSTreeNode(next_key, None, None, None, None)

        if next_key < node.key:
            node.low = self._set(node.low, keys, value)
        elif next_key == node.key:
            if len(keys) > 1:
                node.eq = self._set(node.eq, keys[1:], value)
            else:
                # what happens if you DO NOT add the value here?
                node.value = value
        else:
            node.high = self._set(node.high, keys, value)

        return node

    def set(self, key, value):
        keys = [x for x in key]
        self.root = self._set(self.root, keys, value)
```

你需要使用你学到的“代码大师复制”方法学习。要特别注意如何处理 `node.eq` 路径以及如何设置 `node.value` 。一旦你了解了 `get` 和 `set` 的工作方式，你将实现剩下的函数和所有的测试。要实现的函数有：

> `find_shortest`
>
> 给定一个关键字`K`，找到以`K`开头的最短键/值对。这意味着如果你的`set`中有`apple`和`application`，那么调用`find_shortest("appl")`将返回关联`apple`的值。
>
> `find_longest`
>
> 给定一个关键字`K`，找到以`K`开头的最长键/值对。这意味着如果你的`set`中有`apple`和`application`，那么调用`find_shortest("appl")`将返回关联`application`的值。
>
> `find_all`
>
> 给定一个关键字`K`，找到以`K`开头的所有键/值对。我会先实现它，然后基于它实现`find_shortest`和`find_longest`。
>
> `find_part`
>
> 给定一个关键字`K`，找到最短的键，它拥有`K`的开头的一部分。研究如何以及在哪里设置`node.value`来使其生效。

研究性学习

- 查看原始代码的注释，看看在`_set`过程中，在哪里放置`value`。修改它会修改`get`的含义吗？为什么？
- 确保你使用随机数据来测试，并测量一些性能。
- 你也可以在`TSTree`中进行模糊匹配。我认为这是一个附加题，所以尝试实现它们，看看你想出了什么。模糊匹配是，`'a.p.e'`匹配`"apple"`、`"anpxe"`和`"ajpqe"`。
- 如何搜索字符串的结尾？提示：不要过度考虑它。

练习 24：URL 快速路由

> 原文：Exercise 24: Fast URL Search
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

我们将结束数据结构和算法的部分，并将数据结构用于实际问题。我已经写了几个 Web 服务器，一个不断出现的问题是，将 URL 路径匹配到“动作”。你会在每个 Web 框架，Web 服务器，和必须基于层次化的键来“路由”信息的任何东西中发现此问题。当你的 Web 服务器收到 URL `/do/this/stuff/` 时，必须确定每个部分是否可能附加了某种操作或配置。如果你在 `/do/` 配置了 Web 应用程序，那么你的网络服务器应该使用 `/this/stuff/` 做什么呢？是否认为它是失败的，或将其传递给 Web 应用程序？如果 `/do/this/` 中有一个目录怎么办？而且，如何快速检测到错误的 URL，因此你不必处理不存在的巨大请求？

这种层次化的搜索经常出现，这是对你将算法和数据结构应用于问题的能力，以及性能分析能力进行测试的最佳测试。

挑战练习

首先，请确定你了解 URL 是什么以及如何使用。如果没有，那么我建议你花时间去写一个带有一些复杂路由的小型 Flask 应用程序。这是你将要实现的路由。

接下来，你应该执行以下操作：

- 创建一个简单的基本的 `URLRouter` 类，你将为所有实现派生它。你应该可以对此 `URLRouter` 执行以下操作：
    - 添加一个带有关联对象的新 URL。
    - 获取 URL 的完全匹配。搜索 `/DO/THIS/STUFF/` 只返回正好是它的东西。
    - 获取 URL 的最佳匹配。搜索 `/DO/THIS/STUFF/` 将匹配 `/DO/` ，如果这是唯一的匹配。
    - 获取以此 URL 开头的所有对象。
    - 获取 URL 的最短匹配对象。搜索 `/DO/THIS/STUFF/` 会返回 `/DO/` 而不是 `/DO/THIS/` 。
    - 获取 URL 的最长匹配对象。搜索 `/DO/THIS/STUFF/` 将返回 `/DO/THIS/` 而不是 `/DO/` 。
- 使用 `TSTree` 创建 `URLRouter` 的子类，因为这样最容易了。确保测试了下面这些事情：
    - 不同长度的随机 URL 和路径，在 `TSTREE` 和你搜索的内容里面。
    - 在不同情况下只寻找部分路径
    - 完全不存在的路径
- 存在和不存在的非常长的路径

- 一旦你让这个子类工作，并测试完毕，推广你的测试，所以你可以在所有打算完成的实现中运行它。
- 然后，尝试使用`DoubleLinkedList`，`BSTree`，`Dictionary`和 Python 的`dict`来实现。确保你的泛用测试适用于所有这些。
- 一旦完成了，开始分析这些实现的不同操作的性能。

目标是看看与其他数据结构相比，`TSTree`有多快。它可能会击败大多数东西，但也许 Python `dict`多数情况会赢，因为它针对 Python 进行了优化。你甚至可以为每个操作猜测，哪个数据结构具有最佳性能。

研究性学习

- 我省略了`SuffixArray`，因为它类似于`TSTree`，但为了使用它，你必须添加相同的操作。实现它，然后看看`SuffixArray`如何比较。
- 研究你最喜欢的 Web 服务器或 Web 框架是如何实现的。你会发现很多使用 URL 人不知道什么是三叉搜索树，尽管它对于常见操作非常有用。

深入学习

如果你想深入了解算法和数据结构，我强烈推荐 Steven S. Skiena 的《The Algorithm Design Manual》一书。他的书使用 C，所以你可能需要先阅读《笨办法学 C》，以便能够浏览它。除此之外，它是一本很好的书，因为它涵盖了分析算法和数据结构的性能的理论和实现。

第四部分：进阶项目

> 原文：Part IV: Intermediate Projects
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在第三部分中，你学习了数据结构和算法的基础知识，但更重要的是，你学习了审计和测试代码。你并没有审计和测试你自己的代码。你刚刚通过我教给你的方式，审计了我的缺陷。第四部分的目标是通过一系列挑战模式的项目，审计你自己的代码。在接下来的五个项目中你的任务如下：

- 进行45分钟的 Hack 会话，创建项目并开始。
- 使用你在第三部分中学到的第一个 Hack，审计你的实现中潜在的缺陷和问题。
- 然后在另一个 45 分钟内开始清理，并把你的黑魔法开发成正式的东西。
- 在 45 分钟的会话内审计并优化它。

这 45 分钟的会话与你的第一批项目之间的唯一区别是，你不需要严格限制时间。45 分钟只是一个指导，来确保你不要太久才审计你的代码。审计停留在好的实现或者想法中间的代码是没有意义的。显然，这些半成品代码不值得很好地审计。关键是要工作大约 45 分钟，当你暂停了一段时间，然后查看你做了什么。

在本节中，你将参考第三部分的清单，并严格遵守它。在进行审计之前，请先休息 10~15 分钟，来唤醒头脑并切换到批判的思维模式，这是很好的。

当你处理这些项目时，我将提出一些算法，当它们适合时，你可以在应用程序中使用。你不必使用你实现的算法，但你应该尝试，只是为了看看它们的工作原理。很可能他们不比 Python 现有的数据结构（`list` 和 `dict`）更好，因为 Python 的数据结构已经有了很多调整，来变得尽可能快。尝试使用算法，以便你了解何时使用它们以及如何检查它们，这仍然是一个很好的练习。

记录你的缺陷

最后，我要求你跟踪你的缺陷率。就像在第二部分中，跟踪你完成的功能那样。你将跟踪你在审计中找到了多少缺陷，以及它们有什么样的缺陷。通过创建一个表格，顶部是缺陷类型，左侧是日期时间，在你的日志中记录你发现的东西。如果你想要使用电子表格，你也可以直接绘制结果。跟踪发现这些缺陷的目标是，开始了解你在编程会话期间经常犯的错误，以便你可以尝试阻止他们或在审计中简单留意它们。

练习 25： `xargs`

> 原文：Exercise 25: xargs
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

我们回到挑战模式的练习，并为你热身，你将会实现 `xargs` 。这应该是一个简单的实现，但是 `xargs` 可能很复杂，因为你需要启动其他程序才能使其运行。你要研究的 Python 模块是 `subprocess` ，可以从 Python 运行其他程序并收集其输出。你将需要了解该模块，稍后完成 `xargs` 和本书的许多其他项目，因此要好好研究。

挑战练习

实现 `xargs` 只需 45 分钟，所以你可以做出一些东西，之后你可以审计它。记住第一个 Hack 就是让项目能运行，而不是使其完美。你将在此项目中的后续步骤中优化它并使其更好。记住你可以键入：

```
man xargs
```

获取 `xargs` 的手册页并研究如何工作。这是一个方便的 Unix 工具，但你也可以使用 `find` 做几乎相同的事情。当你实现 `xargs` 时，尝试找出，它比起 `find --exec` 有什么优势。

经过 45 分钟的 Hack，你应该休息一下，然后使用第三部分的代码审计检查清单，对代码进行客观的审计。不要修复代码，只需编写注释，指出需要改变什么，有什么缺陷。在尝试修复时，很难保持客观，所以只需要注意审计中的问题，然后在下一轮中修复它们。

然后，你将进行一系列代码/审计的计时会话，来习惯于进行审计。花费你所需的尽可能长的时间，尽可能多地实现 `xargs` ，然后继续下一个项目。

> 注
>
> 记住要在日志中跟踪你的缺陷，所以你可以绘制它们的运行图，并寻找趋势。

研究性学习

- 在代码/审计的流程中，你是否发现任何你不断犯下的错误？把这些当成潜在的事情写下来并处理。

- 你的代码/审计流程中，是否有一个特定的时间点，有或多或少的缺陷？比起最开始更多，还是三到四个流程之后更多？为什么会这样？
- 尝试为你的`xargs`的实现编写自动测试，并查看是否降低你的缺陷率。在下一个练习中，你将会进行一个更加受控的测试研究，就像这样，但是现在尝试一下，看看你发现了什么。

练习 26： `hexdump`

> 原文：Exercise 26: hexdump
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你已经用 `xargs` 完成了热身，现在正在代码/审计的循环中。你现在将尝试以“测试优先”方式完成下一个挑战。这就是，你编写测试，它描述你的预期行为，然后实现该行为，直到通过测试。你将要复制 `hexdump` 工具，并尝试将你的版本的输出与真实版本匹配。这是“测试优先”开发真正有帮助的地方，因为它自动化了模仿另一个软件的流程。

当你需要编写一个糟糕的软件的替代品时，这种技术非常有用。软件中的一个常见工作是处理一个项目，它的目的是使用更新的实现替换旧系统。一个例子是用一个新的、热门的 Django 系统来替换旧的 COBOL 银行系统。动机通常是，通过使用比旧系统更容易使用的东西，来使其更容易维护和扩展。如果你可以编写一组自动测试来验证旧系统的行为，然后将该测试套件用于新系统，那么你可以通过一种方法，来确认你的替换品几乎正常。相信我，这些替代工作几乎是不可能的，通常不会成功，但自动测试是有帮助的。

这个练习中，你会向你的流程添加下面这些：

- 在你需要实现的场景中，编写一个测试用例，运行原始的 `hexdump` 。让我们假设 `-C` 选项。你将需要使用 `subprocess` 启动它，或者简单地提前运行它，并将结果保存到加载的文件。
- 通过测试你的 `hexdump` 版本，然后比较结果，编写使测试工作的代码。如果他们不等价，那么你就做错了。
- 然后审计测试代码和你的代码。

我选择了 `hexdump` ，因为难度在于，复制其奇怪的输出格式来查看二进制数据。它的工作方式不是特别复杂。它只是匹配你需要的正确输出。这有助于你练习“测试优先”的测试。

> 注
>
> 当我说“先写一个测试”时，我的意思并不是一个庞大的 `test.py` 文件，它具有所有的函数和大量的虚构代码。我的意思是我以前教过的东西。编写一个小型测试用例 - 也许只是一个测试函数的1/10，然后编写代码使其正常工作，然后在两者之间来回跳动。你越了解代码，你就可以写出越多的测试用例，但不要写一堆测试代码，并没有东西来运行它。而是要逐步编写。

挑战练习

当你想要查看不是可见文本的文件内容时，`hexdump`命令很有用。它以各种有用的格式显示文件中的字节，包括十六进制，八进制，并且后面带有 ASCII 输出。实现自己的`hexdump`的难度不是读取数据，甚至不是将其转换为不同的格式。你可以使用 Python 中的`hex`，`oct`，`int`和`ord`函数轻松地执行此操作。原始的格式化字符串运算符也很有用，因为它为固定精度的八进制和十六进制格式化提供了选项。

真正的困难在于为每个不同的选项正确格式化输出，以便它能够正确流动并适合屏幕。以下是Python .pyc文件的hexdump -C输出的前几行：

真正的困难在于为每个不同的选项正确格式化输出，以便它能够正确打印并适合屏幕。以下是`Python .pyc`文件的`hexdump -C`输出的前几行：

```
00000000  03 f3 0d 0a f0 b5 69 57  63 00 00 00 00 00 00 00  |......iWc.......|
00000010  00 03 00 00 00 40 00 00  00 73 3a 00 00 00 64 00  |.....@...s:...d.|
00000020  00 64 01 00 6c 00 00 6d  01 00 5a 01 00 01 64 00  |.d..l..m..Z...d.|
00000030  00 64 02 00 6c 02 00 6d  03 00 5a 03 00 01 64 03  |.d..l..m..Z...d.|
00000040  00 65 01 00 66 01 00 64  04 00 84 00 00 83 00 00  |.e..f..d........|
```

这个“规范”格式化的手册页说：

- 以十六进制显示输入偏移量。所以 10 不是十进制中的 10，它是十六进制。你知道十六进制吗？
- 十六个空格分隔的，两列十六进制字节。这是转换为十六进制的每个字节。多少列代表一个字节？
- 然后以`%_p`格式显示相同的十六个字节，看起来像 Python 格式化占位符，但它专用于`hexdump`。你需要阅读更多手册页，来了解其含义。

之后`hexdump`也可以从`stdin`输入接收输入，这意味着你可以将东西使用管道连接到它：

```
echo "Hello There" | hexdump -C
```

这会在我的 macOS 上产生如下输出：

```
00000000  48 65 6c 6c 6f 20 54 68  65 72 65 0a              |Hello There.|
0000000c
```

请注意，最后一行有一个字符`c`？猜猜看这是什么。

这就是格式化和输出，它比较困难，你的任务是尽可能复制它，这就是为什么这个练习的开头让你以“测试优先”的方式工作。创建测试，将你的数据扔给`hexdump`将会更容易，并将其与真正的`hexdump`进行比较，直到它开始工作。

研究性学习

研究`od`命令，看看你的`hexdump`代码是否可以复用于`od`的实现。如果可以的话，可以制作一个他们都使用的库。

深入学习

有人主张只做“测试优先”的开发，但我相信没有永远适用的技术。当我从用户的角度测试软件的交互时，我更喜欢写测试。我将编写测试，它描述了用户与软件的交互，然后实现软件。这是你所做的事情，因为你正在测试，用户如何从你的`hexdump`命令行调用中看到输出。

对于其他类型的编程任务，决定首先写测试还是编写代码是荒谬的，只会扼杀你解决问题的能力。自动化测试是简单的工具，你是一个聪明的人，有权力尝试使用工具，但你认为他们将在每种情况下都能最好地工作。任何告诉你区别的人可能是一个无理取闹的人，实际上并不擅长编程。

练习 27： `tr`

> 原文：Exercise 27: tr
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

这个练习是继续学习 TDD（又称“测试优先”）风格的开发。知道如何这样编程很重要，因为它在许多地方使用，但如前所述，它有其局限性。在实现 `tr` 命令时，您将再次使用 TDD 进行练习。要十分确定，你是先严格编写测试，然后是代码，然后再审计两个东西。

在上一个练习中，我让你逐步构建测试用例和代码。这通常是最不容易出错的开发方法，但它不能帮助您更好地分析自己的代码。在这个练习中，你会做一些略微不同的事情，因为我将会写一个完整的测试用例，进行审计，然后编写整个代码，进行审计，并通过运行测试来确认审计。

这意味着，在这个练习中你的流程是这样的：

- 尝试编写大部分 TDD 测试用例。
- 审计测试用例并确认它是否编写正确。
- 运行测试以确保它们失败，但是查找任何语法错误。此时您不应该有语法错误。
- 为测试用例编写代码，但不要运行测试。
- 审计您的代码，并尝试在运行测试之前查看有多少缺陷。

您将在下一个练习中，使用此过程来跟踪您的审计技能、测试技能的指标，并更好地控制您编写代码的方式。

挑战练习

`tr` 工具是翻译字符流的有效方式。尽管非常简单，它可以对字符做一些非常复杂的事情。例如，您可以使用 `tr` ，以一行代码，获取 `history` 中使用的单词的频率：

```
history | tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c | sort -rn
```

似乎很炫酷，但是 Doug McIlroy 曾经使用这一行来辩称，高德纳（Donald Knuth）编写的一个类似的程序太长。Knuth 的实现是“10页”，从头开始构建一切。Doug 的一行只是使用标准的 Unix 工具来做同样的事情。这展示了 Unix 的管道工具的力量和 `tr` 的翻译文本的能力。

使用手册页和任何其他东西，弄清 `tr` 命令的作用。还有一个同名的 Python 项目，但是我会告诉你要避开它，直到你完成实现，所以你可以稍后再比较这个项目。同时不要忘记，为此你需要一个整体的项目，它应该是测试完成的 TDD 风格，就像我开始的描述的那样。

45 分钟的批判

我希望你继续使用45分钟的时间，但是有一个对这种工作方式的很大批评：你不能进入扩展的专注流程。在短时间内工作，像这样，在你需要处理大量工作的，以及必须加快步伐时有所帮助。这种情况发生在工作真的很无聊，没有乐趣的时候。我正在让你使用45分钟的时间块来加快自己的速度，但是我们也会在稍后使用它们，来收集一些指标，有关如何在时间中工作，来进行后续分析。

但我会提醒你，最好的编程是在专注的状态中完成的。这就是，你的注意力在几个小时之内高度集中，失去所有的时间感，直到凌晨 5 点，意识到你度过了一整夜。这种强烈的专注使编程对我来说非常愉快，但是当您对您正在做的事情很感兴趣时，它真的是可持续的。当您需要处理别人的糟糕的代码库时，这个现象往往不会发生。在这些情况下，您需要一个不同的策略，加快你的工作并让你摆脱困境，而不会损耗你的兴致。这就是 45 分钟的时间块的作用。

最后，建立你进入状态并集中精力几个小时的能力，一个方法是从短暂的时间开始，然后慢慢增加它们，直到你可以忍受更长的时间。继续使用 45 分钟的时间快，但是如果你只是忘乎所以，最后在最后几个小时内完成黑魔法，那么就玩的开心。没有人会说你做错了，这实际上是正常的。

研究性学习

- 这种工作方式怎么样？你喜欢吗？尝试阐明为什么，然后阅读一些当前的 TDD 的文章，或它的近亲行为驱动开发（BDD）。
- 你认为通过首先审计你的代码而不是逐步构建它，你发现了更多还是更少的缺陷？猜测它，然后写下来。

练习 28： `sh`

> 原文：Exercise 28: sh
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你现在将继续你的 TDD 风格流程，但你将添加一个小型的 Hack 会话来起步。使用 TDD 工作的最佳方法，实际上不是首先编写测试，而是以这种方式工作：

- 花45分钟的时间来研究这个问题。这被称为“spike”，旨在解决你可能遇到的问题或学习你需要知道的事情。
- 使用 TODO 列表来计划你可能需要实现什么。
- 将此计划变成 TDD 测试。
- 运行测试来确保它失败。
- 编写测试代码，使用你从 spike 学到的东西。
- 审计并测试你的代码来确认质量。

当 TDD 狂热者遇到从没学过的问题时，这个过程是我看到它们实际使用的东西。快速实现一个黑魔法，让你的思维活跃，并研究问题，然后认真对待工作更加实际。如果有人告诉你这不是 TDD，只是不要告诉他们你实现做了 spike 。他们永远不会知道。

挑战练习

在本练习中，你将实现 Unix `sh` 工具的 shell 部分。你在编码时一直使用 `sh` ，因为它在终端内部运行（PowerShell 不一样），并运行其他程序。通常它是 `bash` ，但它可能是 `fish` ， `csh` 或 `zsh` 。

`sh` 工具是一个需要实现的庞大的程序，因为它也支持一个完整的编程语言，来自动化你的系统。我们不会实现编程语言，只是命令行进程运行的那部分。

要完成此任务，你需要以下库：

- `subprocess` ，启动其他程序。
- `readline` ，从用户获取输入和支持历史记录。

你不用做一个带管道和所有东西的完整的 Unix `sh` ，但是应该实现除编程语言之外的所有东西。你的实现应该能够执行以下操作：

- 使用 `readline` ，从提示开始，并从用户获取命令来执行。

- 将命令解析成可执行文件和参数。
- 使用 `subprocess` 执行具有参数的命令，并控制所有的输出。

为了起步，你可以做你的 spike，来学习 `readline` 或 `subprocess` 或两者，任何你认为是必要的或不熟悉的东西。一旦你完成了 spike，那么你开始编写测试和实现系统。

研究性学习

你可以实现管道吗？就是你键入 `history | grep python` ，并且 `|` 将 `history` 的输出发给 `grep` 的输入。

深入学习

如果你打算深入了解 Unix 进程和资源管理，你可以研究我的项目 `python-lust` 。它并不是非常大，并且充满了许多小技巧。

练习 29： `diff` 和 `patch`

> 原文：Exercise 29: diff and patch
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

为了完成第四部分，你将简单把你所学习的完整的 TDD 流程，应用于你可能不熟悉的、更相关的项目。请参阅练习 28 来确认你了解该流程，并确保你严格遵循它。如果必须的话，创建一个检查列表。

> 警告
>
> 当你实际工作时，这个严格的流程完全没有用。目前，你正在研究该流程，并将其内在化，以便你可以在现实世界中使用它。这就是为什么我让你严格遵循它。这只是练习，所以当你做真正的工作时，不要成为一个狂热者。这本书的目的是，教你一套完成工作的策略，而不是教你一个可以传播给大众的宗教仪式。

挑战练习

`diff` 命令接受两个文件并产生第三个文件（或输出），它包含第一个文件与第二个文件相比，修改的东西。它是 `git` 和其它版本控制工具的基础。在 Python 中实现 `diff` 是相当简单，因为有一个库可以为你做这件事，所以你不需要处理算法（这可能非常复杂）。

`patch` 工具是 `diff` 工具的伙伴，因为它需要一个差异文件，并将其应用到另一个文件，来产生第三个文件。这可以让你选取在两个文件中的更改，运行 `diff` 来仅仅生成差异，然后将该 `.diff` 文件发送给某人。那个人可以使用他们的原始文件副本和 `.diff` ，使用 `patch` 来重建你的更改。

以下是一个工作流程示例，来演示 `diff` 和 `patch` 的工作原理。我有两个文件 `A.txt` 和 `B.txt` 。 `A.txt文件包含一些简单的文字，然后我复制它，并创建` B.txt`，带有一些修改：

```
$ diff A.txt B.txt > AB.diff
$ cat AB.diff
2,4c2,4
< her fleece was white a mud
< and every where that marry
< her lamb would chew cud
---
> her fleece was white a snow
> and every where that marry went
> her lamb was sure to go
```

这产生了文件`AB.diff`，它拥有`A.txt`与`B.txt`相比的变化，你可以看到这是在修复我打破的押韵。一旦你有了`AB.diff`，你可以使用补丁应用更改：

```
$ patch A.txt AB.diff
$ diff A.txt B.txt
```

最后的命令应该不显示认识输出，因为之前的`patch`命令使`A.txt`与`B.txt`具有相同的内容。

这两个东西的实现，应该从`diff`命令开始，因为使用 Python 来作弊，你有完全实现的`diff`。你可以在`difflib`文档的末尾找到它，但尝试实现你的版本，并看看与之相比怎么样。

这个练习的真正要点就是`patch`工具，Python 没有为你实现它。你将要阅读`difflib`中的`SequenceMatcher`类，并特别查看`SequenceMatch.get_opcodes`函数。这是你`patch`工作的唯一线索，但这是一个非常好的线索。

研究性学习

你能把这种`diff`和`patch`的组合做到什么程度？你可以将它们组合成一个工具吗？你可以让他们像微型的`git`那样工作吗？

深入学习

找到尽可能多的差异比较算法。另一件需要研究的事情是`git`的工作方式。

第五部分：文本解析

> 原文：Part V: Parsing Text
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

本书的这一部分将教你如何处理文本，特别是，它是文本解析的正式开始。所以我不会涉及编程语言理论的所有不同理论元素，因为这是整个大学的学位。这只是简单而朴素的文本解析的开始，可以在许多编程环境中使用它。

大多数程序员与解析文本有着奇怪的关系。所有计算机程序设计的核心是解析，它是计算机科学中最容易理解和形式化的方向之一。解析数据在计算中无处不在。你可以在网络协议，编译器，电子表格，服务器，文本编辑器，图形渲染器，以及拥有人机或其他计算机接口的任何东西中找到它。即使两台计算机正在发送固定的二进制协议，尽管缺少文本，仍然存在解析的层面。

我要教你解析，因为它是一种容易理解的可靠技术，可以产生可靠的结果。当你面对可靠地处理一些输入并给出准确的错误时，你将求助于解析器，而不是手动编写一个。另外，一旦学习了解析的基础，就会更容易学习新的编程语言，因为你可以理解他们的语法。

代码覆盖简介

在这部分中，你仍然应该尝试拆解和剖析你编写的任何代码。我在这部分中增加的新东西，是代码覆盖的概念。代码覆盖的想法是，你实际上不知道在编写自动测试时是否测试了大多数情况。你可以使用形式逻辑来开发一个理论，即你覆盖了一切东西，但是我们知道人类的大脑非常难以在自己的思维中找到缺陷。这就是为什么你在这本书中使用“创造然后批判”的循环。在尝试创建某些东西的时候，你很难分析自己的想法。

代码覆盖是一种方法，至少能够了解你在应用中测试的东西。它不会找到你所有的缺陷，但它至少会显示，你已经命中每个可能的代码分支。如果没有覆盖，你实际上不知道你是否测试了每个分支。一个非常好的例子是故障处理。大多数自动测试仅测试最可靠的条件，并且不会测试错误处理。当你运行覆盖时，你会发现你忘记的所有方法，来测试错误处理代码。

代码覆盖也可以帮助你避免过度测试代码。我曾经从事测试驱动开发（TDD）爱好者的项目，他们因 12/1 的测试/代码比而感到自豪（这意味着每一行代码都有 12 行测试）。一个简单的代码覆盖分析显示，他们只测试了 30% 的代码，其中许多线路以同样的方式进行了

6~20 次的测试。同时，像数据库查询中的异常情况那样的简单错误是完全未经测试的，并导致频繁的错误。最终，这些测试套件成为一种负担，阻止了项目的成长，并且只会吞掉开发人员的工作安排。难怪这么多敏捷咨询公司讨厌代码覆盖。

在本练习的视频中，你将看到我运行测试，并使用代码覆盖来确认我正在测试什么。我要求你做同样的事情，并且有使其变得容易的工具。我将向你展示如何阅读测试覆盖结果，以及如何确保你高效地测试你可以测试的一切东西。目标是拥有一个彻底的自动化测试套件，但不会浪费你的努力，所以你不会连续测试 12 次只有 30% 的代码。

练习 30：有限状态机

> 原文：Exercise 30: Finite State Machines
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

每当你阅读一本关于解析的书，都有一个可怕的章节，关于有限状态机（FSM）。他们对“边”和“节点”进行了详细的分析，每个可能的“自动机”的组合被转换成其他自动机，坦率地说，它有点多了。FSM 有一个更简单的解释，使得它们实用并且可理解，而不会违背相同主题的纯理论版本。当然你不会向 ACM 提交论文，因为你不知道 FSM 背后的所有数学知识，但如果你只想在应用程序中使用它们，那么它们就足够简单了。

FSM 是组织事件一种方式，事件发生在一系列状态上。定义事件的另一种方法是“输入触发器”，类似于 `if` 语句中的布尔表达式，但通常不太复杂。事件可以是按钮点击，从流中接收字符，更改日期或时间，以及几乎任何用于声明事件的东西。状态就是你的 FSM 停止的任何“位置”，同时它等待更多的事件，并且你定义的每个状态都允许事件（输入）。事件往往是暂时的，而状态通常是固定的，而且二者都是可以存储的数据。最后，你可以将代码附加到事件或状态，甚至决定在进入状态时，状态中或退出状态时是否应运行代码。

FSM 只是一种方法，在执行中不同位置发生不同事件时，使用白名单列出可能运行的代码。在 FSM 中，当你收到意外事件时，你会发生故障，因为你必须明确说明每个状态允许哪些事件（或条件）。`if` 语句也可以处理可能的分支，但它是一个可能性的黑名单。如果你忘记了 `else` 子句，那么你的 `if-elif` 条件没有覆盖的任何东西都会退回默认。

让我们将其拆解：

- 你拥有状态，这是 FSM 当前所在位置的存储指示器。状态可以是“开始”，“按下某键”，“中止”或类似的方式，描述执行的可能位置中的 FSM 的位置。每个状态都意味着正在等待某事发生，在决定下一步做什么之前。
- 你拥有事件，可以将 FSM 从一个状态移动到另一个状态。事件可以是“按下某键”，“套接字连接失败”，“文件保存”，并表示 FSM 接收到一些外部刺激，因此必须决定要做什么，以及下一个状态是什么。一个事件甚至可以回到同一个状态，这是你循环的方式。
- 根据发生的事件，FSM 从一个状态转换到另一个状态，并且仅仅由于为状态提供的确切事件（尽管其中一个事件可以定义为“任何事件”）。他们不会“意外”转移状态，你可以通过查看收到的事件和访问的状态，精确地跟踪他们从一个状态转移到另一个状态。这使得它们非常容易调试。
- 在状态转换之前，之后和期间，你可以在每个事件上运行代码。这意味着你可以在收到事件时运行一些代码，然后决定在该状态下基于该事件做什么，然后在离开该状态之前

再次运行代码。这种执行代码的功能使得 FSM 非常强大。

- 有时候“没有”也是一个事件。这很好很强大，因为这意味着即使没有发生任何事情，你也可以将 FSM 转换到新的状态。然而，实际上，“没有”往往是隐含的事件“再来一次”或“醒来”。在其他情况下，这个状态的意思是，“不确定，也许下一个事件会告诉我是什么状态。”

FSM 的力量是能够明确地说明每个事件，事件只是正在接收的数据。这使得它们非常容易进行调试，测试和正确实现，因为你确切地知道每个状态的可能性，以及在每个状态中，对于每个事件可能发生的情况。在本练习中，你将要研究 FSM 库和使用它的 FSM 实现，来了解它们如何工作。

挑战练习

我创建了一个 FSM 模块，处理一些简单的事件来处理 Web 服务器的连接。这是一个虚构的 FSM，为你提供一个在 Python 中快速编写 FSM 的例子。它只是处理连接的基本框架，连接从套接字读取和写入，并且它缺少一些重要的东西，但这只是供你使用的一个很小的例子。

```python
def START():
    return LISTENING

def LISTENING(event):
    if event == "connect":
        return CONNECTED
    elif event == "error":
        return LISTENING
    else:
        return ERROR

def CONNECTED(event):
    if event == "accept":
        return ACCEPTED
    elif event == "close":
        return CLOSED
    else:
        return ERROR

def ACCEPTED(event):
    if event == "close":
        return CLOSED
    elif event == "read":
        return READING(event)
    elif event == "write":
        return WRITING(event)
    else:
        return ERROR

def READING(event):
    if event == "read":
        return READING
    elif event == "write":
        return WRITING(event)
    elif event == "close":
        return CLOSED
    else:
        return ERROR

def WRITING(event):
    if event == "read":
        return READING(event)
    elif event == "write":
        return WRITING
    elif event == "close":
        return CLOSED
    else:
        return ERROR

def CLOSED(event):
    return LISTENING(event)

def ERROR(event):
    return ERROR
```

也有一个小测试，向你展示如何运行这个 FSM：

```python
import fsm

def test_basic_connection():
    state = fsm.START()
    script = ["connect", "accept", "read", "read", "write", "close", "connect"]

    for event in script:
        print(event, ">>>", state)
        state = state(event)
```

你在本练习中的挑战是，将此示例模块变成一个更强大和通用的 FSM Python 类。你应该使用它作为一系列线索，来了解如何处理进入的事件，状态如何作为 Python 函数，以及如何进行隐式的转换。看看我有时候为下一个状态返回函数，但其他时候我会返回一个状态函数的调用？试着弄清楚为什么我会这样做，因为它在 FSM 中非常重要。

为了完成这个挑战，你需要学习 Python `inspect` 模块，看看你可以用 Python 对象和类来做什么。有一些特殊的变量，如 `__dict__` 以及 `inspect` 中的函数，可帮助你窥探类或对象并查找函数。

你也可以决定要反转此设计。你可以将事件作为子类中的函数，并在事件函数内检查当前的 `self.state`，来确定接下来要执行的操作。这完全都取决于你正在处理什么，你是否拥有更多的事件还是状态，当时什么有意义。

最后，你可以使用一个设计，其中有一个 `FSMRunner` 类，它只知道如何运行这样设计的模块。这比一个知道如何运行自身实例的单一类有一些优点，但也有一些问题。例如，`FSMRunner` 如何跟踪当前状态？它放在模块中还是在 `FSMRunner` 的实例中？

研究性学习

- 使你的测试更加泛用，并为你熟悉的完全不同的领域做一个FSM。
- 添加一个功能，启动在你的实现中运行的事件的日志。使用 FSM 处理事件的最大优点之一是，可以存储和记录 FSM 收到的所有事件和状态。这可以让你调试，为什么它达到你不需要的状态。

深入学习

你应该仔细研究 FSM 背后的数学。我这里的小例子不是完全形式化的概念版本，以便你能理解它。

练习 31：正则表达式

> 原文：Exercise 31: Regular Expressions
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

正则表达式（RegEx）是一种简洁的方式，用于确定字符序列应如何在字符串中匹配。通常大家都认为它们是“可怕”的，但是，正如你所知道的，任何包含在恐惧中的东西通常都不是这样。正则表达式的事实是，它们是大约八个符号的集合，告诉计算机如何匹配模式串。简单来说，他们很容易理解。人们遇到困难的地方是，尝试使用难以置信的复杂的正则表达式，其中解析器实际上会更好。一旦你明白了这八个符号和正则表达式的限制，你就会看到它们根本不可怕。

我打算让你记忆更多东西，使你的的大脑为讨论做好准备。

> `^`
>
> 锚定字符串开头。只有字符串刚好位于开头，它才会匹配。
>
> `$`
>
> 锚定字符串末尾。只有字符串到达了末尾，它才会匹配。
>
> `.`
>
> 任何单个字符。接受任何单个字符的输入。
>
> `?`
>
> 正则表达式的之前的部分是可选的，所以 `A?` 的意思是可选的字符 `A`。
>
> `*`
>
> 之前的部分是零个或多个（任意个）。选取正则表达式的之前的部分，重复接受或者跳过它。`A*` 会接受 `"AAAAAAA"` 或者 `"BQEFT"`，因为它里面有零个 `A`。
>
> `+`
>
> 之前的部分是一个或多个（至少一个）。和 `*` 类似，但是只接受一个或多个这种字符。`A+` 会匹配 `"AAAAAAA"`，但不是 `"BQEFT"`。
>
> `[X-Y]`
>
> `X` 到 `Y` 的字符范围，接受任何范围中列出的字符串。`[A-Z]` 表示所有大写英文字母。许多常见字符范围拥有 `\` 快捷方式，你可以使用它来代替。
>
> `()`
>
> 捕获这个正则表达式的部分，便于稍后使用。许多正则表达式库将其用于替换、提取或修改文本。捕获会选取正则表达式的 `()` 中的部分，并保存它便于以后使用。之后许多库可以让你引用这些捕获。如果你使用 `([A-Z]+)`，它会捕获一个或多个大写英文单词。

Python 的 `re` 库列出了一些更多的符号，但大多都是这八个的一些修饰符，或者不在正则表达式库中经常发现的额外功能。你将快速记住这八个来起步，重点是粗体的部分（锚定末尾，之前部分可选），以便你可以快速回忆它们并解释他们的作用。

记住这些符号后，请查看以下正则表达式并将其翻译成中文，并使用 Python `re` 库来尝试列出的字符串，或你可以想到的任何其他字符串。

> ".*BC?$"
>
> helloBC , helloB , helloA , helloBCX
>
> "[A-Za-z][0-9]+"
>
> A1232344 , abc1234 , 12345 , b493034
>
> "^[0-9]?a*b?.$"
>
> 0aaaax , aaab9 , 9x , 88aabb , 9zzzz
>
> "A+B+C+[xyz]*"
>
> AAAABBCCCCCCxyxyz , ABBBBCCCxxxx , ABABABxxxx

一旦你翻译了它们，使用Python `re` 模块，尝试在 Shell 中尝试它们，如下：

```
>>> import re
>>> m = re.match(r".*BC?$", "helloB").span()
>>> re.match(r".*BC?$", "helloB").span()
(0, 6)
>>> re.match(r"[A-Za-z][0-9]+", "A1232344").span()
(0, 8)
>>> re.match(r"[A-Za-z][0-9]+", "abc1234").span()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
>>> re.match(r"[A-Za-z][0-9]+", "1234").span()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
>>> re.match(r"[A-Za-z][0-9]+", "b493034").span()
(0, 7)
>>>
```

对于任何不匹配，你会得到 `AttributeError: 'NoneType'` ，因为当你的正则表达式不匹配时， `re.match` 函数返回 `None` 。

挑战练习

挑战是尝试使用你的 FSM 模块来实现一个简单的正则表达式，至少执行三个操作。这将是一个困难的挑战，但使用 Python `re` 库来帮助你规划和测试此正则表达式的实现。然后，一旦你知道如何实现它，永远不要这样做了。人生苦短，不要做计算机已经擅长的事情。

研究性学习

- 扩展你的记忆，来包括 Python `re` 库文档中的所有可能的符号。
- 如果你想要匹配一个 `*` 字符，那么你可以用 `\*` 来转义它。大多数其他符号也有类似的东西。
- 确保你知道如何使用 `re.ASCII` ，因为某些解析的需求需要它。

深入学习

看看 `regex` 库，如果你需要 Unicode 支持，那么这个更好。

练习 32：扫描器

> 原文：Exercise 32: Scanners
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

我的第一本书在练习 48 中非常偶然涉及到了扫描器，但现在我们将会更加正式。我将解释扫描文本背后的概念，它与正则表达式有关，以及如何为一小段 Python 代码创建一个小型扫描器。

我们以下面的 Python 代码为例来开始讨论：

```
def hello(x, y):
    print(x + y)

hello(10, 20)
```

你已经在 Python 上练习了一段时间了，所以你的大脑最有可能很快阅读这个代码，但是你真的明白了吗？当我（或别人）教你 Python 时，我让你记得所有的“符号”。`def` 和 `()` 字符是每一个符号，但是 Python 需要一种可靠的、一致的方法来处理它们。Python 还需要能够读取 `hello`，理解它是一个什么东西的“名称”，然后知道 `def hello(x, y)` 和 `hello(10, 20)` 之间的区别。怎么实现它呢？

执行此操作的第一步是，扫描文本并查找“记号”（Token）。在扫描阶段，像 Python 这样的语言不会首先关心什么是符号（`def`），什么是名称（`hello`）。它将简单地，尝试将输入语言转换为的文本模式串，成为“记号”。它通过应用一系列正则表达式来做到这一点，这些正则表达式“匹配” Python 理解的每个可能的输入。练习 31 中，你会记得一个正则表达式是一种方式，告诉 Python 要匹配或接受什么字符序列。所有 Python 解释器都使用许多正则表达式，来匹配它理解的每个记号。

如果你看看上面的代码，你可以编写一组正则表达式来处理它。`def` 需要一个简单的正则表达式，只是“def”。对于 `()+:,` 字符你需要更多的正则表达式。然后，你还剩下如何处理 `print`，`hello`，`10` 和 `20`。

一旦你确定了上述代码示例中的所有符号，你需要命名它们。你不能仅仅通过它们的正则表达式来引用它们，因为查找效率低下，也令人困惑。稍后你会发现，为每个符号提供自己的名字（或数字）可以简化解析，但现在让我们为这些正则表达式设计一些名称。我可以说 `def` 只是 `DEF`，那么 `()+:,` 可以是 `LPAREN RPAREN PLUS COLON COMMA`。之后，我可以将用于 `hello` 和 `print` 之类的单词正则表达式称为 `NAME`。通过这样做，我想出了一种方法，将原始文本流转换成一个单个数字（或名称）记号的流，来在后期使用。

Python 也很棘手，因为它需要一个前导空白的正则表达式，来处理代码块的缩进和压缩。现在，让我们使用一个相当笨的 `^\s+`，然后假装它也捕捉到行的开头使用了多少个空白。

最终你会拥有一组正则表达式，可以处理上面的代码，它可能看起来像这样：

正则表达式	记号
`def`	`DEF`
`[a-zA-Z_][a-zA-Z0-9_]*`	`NAME`
`[0-9]+`	`INTEGER`
`\(`	`LPAREN`
`\)`	`RPAREN`
`\+`	`PLUS`
`:`	`COLON`
`,`	`COMMA`
`^\s+`	`INDENT`

扫描器的任务是使用这些正则表达式，并将输入文本分解成识别符号的流。如果我这样对示例代码这么做，我可以产生：

```
DEF NAME(hello) LPAREN NAME(x) COMMA NAME(y) RPAREN COLON
INDENT(4) NAME(print) LPAREN NAME(x) PLUS NAME(y) RPAREN
NAME(hello) RPAREN INTEGER(10) COMMA INTEGER(20) RPAREN
```

研究此转换，匹配扫描器输出的每一行，并使用表中的正则表达式将其与上述 Python 代码进行比较。你会看到这只是选取输入文本，将每个正则表达式匹配到记录名称，然后保存所需的任何信息，如 `hello` 或数字 `10` 。

微型 Python 扫描器

我编写了一个微型 Python 扫描器，演示了这个微型 Python 语言的扫描：

```
import re

code = [
"def hello(x, y):",
"    print(x + y)",
"hello(10, 20)",
]

TOKENS = [
(re.compile(r"^def"),                    "DEF"),
(re.compile(r"^[a-zA-Z_][a-zA-Z0-9_]*"), "NAME"),
(re.compile(r"^[0-9]+"),                 "INTEGER"),
(re.compile(r"^\("),                     "LPAREN"),
(re.compile(r"^\)"),                     "RPAREN"),
(re.compile(r"^\+"),                     "PLUS"),
(re.compile(r"^:"),                      "COLON"),
(re.compile(r"^,"),                      "COMMA"),
(re.compile(r"^\s+"),                    "INDENT"),
]

def match(i, line):
    start = line[i:]
    for regex, token in TOKENS:
        match = regex.match(start)
        if match:
            begin, end = match.span()
            return token, start[:end], end
    return None, start, None

script = []

for line in code:
    i = 0
    while i < len(line):
        token, string, end = match(i, line)
        assert token, "Failed to match line %s" % string
        if token:
            i += end
            script.append((token, string, i, end))

print(script)
```

当你运行这个脚本时，你会得到一个 `tuples` 的 `list`，它是 `TOKEN`、匹配到的字符串、开头和末尾，像这样：

```
[('DEF', 'def', 3, 3), ('INDENT', ' ', 4, 1), ('NAME', 'hello', 9, 5),
('LPAREN', '(', 10, 1), ('NAME', 'x', 11, 1), ('COMMA', ',', 12, 1),
('INDENT', ' ', 13, 1), ('NAME', 'y', 14, 1), ('RPAREN', ')', 15, 1),
('COLON', ':', 16, 1), ('INDENT', '    ', 4, 4), ('NAME', 'print', 9, 5),
('LPAREN', '(', 10, 1), ('NAME', 'x', 11, 1), ('INDENT', ' ', 12, 1),
('PLUS', '+', 13, 1), ('INDENT', ' ', 14, 1), ('NAME', 'y', 15, 1),
('RPAREN', ')', 16, 1), ('NAME', 'hello', 5, 5), ('LPAREN', '(', 6, 1),
('INTEGER', '10', 8, 2), ('COMMA', ',', 9, 1), ('INDENT', ' ', 10, 1),
('INTEGER', '20', 12, 2), ('RPAREN', ')', 13, 1)]
```

这个代码绝对不是你可以创建的最快或最准确的扫描器。这是一个简单的脚本，用于演示扫描器的工作原理。对于进行真正的扫描工作，你将使用一种工具来生成更高效的扫描器。我在深入学习部分介绍。

挑战练习

你的工作是研究这个扫描器示例代码，并将其转换成通用的 `Scanner` 类以便稍后使用。这个 `Scanner` 类的目标是接受一个输入文件，将其扫描为记号的列表，然后允许你按顺序取出记号。API 应具有以下功能：

> `__init__`
>
> 使用类似的元组列表（没有 `re.compile`）来配置扫描器。
>
> `scan`
>
> 接受一个字符串并执行扫描，创建一个记录列表以便以后使用。你应该保留这个字符串，让人们以后访问。
>
> `match`
>
> 提供可能的记号列表，返回列表中的第一个记号，并将其移除。
>
> `peek`
>
> 提供可能的记号列表，返回列表中的第一个记号，但不将其移除。
>
> `push`
>
> 将记号放回记号流中，以便后续的 `peek` 或者 `match` 返回它。

你也应该创建通用的 `Token` 类来代替我使用的 `tuple`。它应该能够跟踪发现的记号，匹配的字符串、原始字符串中匹配位置的开头和末尾。

研究性学习

- 安装 `pytest-cov` 库，并使用它来测量自动化测试的覆盖率。
- 使用 `pytest-cov` 的结果来改进自动化测试。

深入学习

创建扫描器的更好方法是，利用以下关于正则表达式的三个事实：

- 正则表达式是有限状态机。
- 你可以将小型有限状态机精确地组合成更大更复杂的有限状态机。
- 匹配许多小型正则表达式的有限状态机组合，操作方式每个正则表达式一样，并且效率更高。

有许多工具使用这个事实来接受扫描器定义，将每个小的正则表达式转换为 FSM，然后将它们组合来产生大段代码，可以可靠地匹配所有记号。这样做的优点是，你可以以滚动方式为这些生成的扫描器提供独立的字符，并使其快速识别记号。它比我这里的方式要好，其中我拼凑字符串，并尝试一系列正则表达式，直到找到一个正则表达式。

研究扫描器的发生器如何工作，并将其与你编写的代码进行比较。

练习 33：解析器

> 原文：Exercise 33: Parsers
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

想象一下，你将获得一个巨大的数字列表，你必须将其输入到电子表格中。一开始，这个巨大的列表只是一个空格分隔的原始数据流。你的大脑会自动在空格处拆分数字流并创建数字。你的大脑像扫描器一样。然后，你将获取每个数字，并将其输入到具有含义的行和列中。你的大脑像一个解析器，通过获取扁平的数字（记号），并将它们变成一个更有意义的行和列的二维网格。你遵循的规则，什么数字进入什么行什么列，是你的“语法”，解析器的工作就是像你对于电子表格那样使用语法。

我们再来看一下练习 32 中的微型 Python 代码，再从三个不同的角度讨论解析器：

```
def hello(x, y):
    print(x + y)

hello(10, 20)
```

当你查看这个代码时，你看到什么？我看到一棵树，类似于我们之前创建的 `BSTree` 或 `TSTree` 。你看到树了吗？我们从这个文件的最上方开始，学习如何将字符转换为树。

首先，当我们加载一个 `.py` 文件时，它只是一个“字符”流 - 实际上是字节，但 Python 使用 Unicode，所以必须处理字符。这些字符在一行中，毫无结构，扫描器的任务是增加第一层次的意义。扫描器通过使用正则表达式，从字符串流中提取意义，创建记号列表。我们已经将一个字符列表转换为一个记号列表，但看看 `def hello(x,y):` 函数。这是一个函数，里面有代码块。这意味着某种形式的“包含”或“东西里面的东西”的结构。

一个很容易表示包含的方式是用一棵树。我们可以使用表格，像你的电子表格一样，但它并不像树那么容易。接下来看看 `hello(x, y)` 部分。我们有一个 `NAME(hello)` 记号，但是我们要抓取 `(...)` 部分的内容，并且知道它在括号内。再次，我们可以使用一个树，我们将 `(...)` 部分中的 `x, y` 部分“嵌套”为树的子节点或分支。最终，我们就拥有了一棵树，从这个 Python 代码的根开始，并且每个代码块， `print` ，函数定义和函数调用都是根的分支，它们也有子分支，以此类推。

为什么我们这样做？我们需要基于其语法，知道 Python 代码的结构，以便我们稍后分析。如果我们不将记号的线性列表转换成树结构，那么我们不知道函数，代码块，分支或表达式的边界在哪里。我们必须以“直线”方式在飞行中确定边界，这不容易使其可靠。很多早期的糟糕

语言是直线语言，我们现在知道了他们不必须是这样。我们可以使用解析器构建树结构。

解析器的任务是从扫描器中获取记号列表，并将其翻译成更有意义的语法树。你可以认为解析器是，对记号流应用另一个正则表达式。扫描器的正则表达式将大量字符放入记号中。解析器的“正则表达式”将这些记号放在盒子里面，它里面有盒子，以此类推，直到记号不再是线性的。

解析器也为这些盒子添加了含义。解析器将简单地删除 `()` 括号记号，并为可能的 `Function` 类创建一个特殊的 `parameters` 列表。它会删除冒号，无用的空格，逗号，任何没有真正意义的记号，并将其转换为更易于处理的嵌套结构。最后的结果可能看起来像，上面的示例代码的伪造树：

```
* root
  * Function
    - name = hello
    - parameters = x, y
    - code:
      * Call
        - name = print
        - parameters =
            * Expression
              - Add
                - a = x
                - b = y
  * Call
    - name = hello
    - parameters = 10, 20
```

递归下降解析

有几种已建立的方法，可以为这种语法创建解析器，但最简单的方法称为递归下降解析器（RDP）。我实际上在我《笨办法学 Python》练习 49 中讲解了这个话题。你创建了一个简单的 RDP 解析器来处理你的小游戏语言，你甚至不了解它。在本练习中，我将对如何编写 RDP 解析器进行更正式的描述，然后让你使用我们上面的 Python 小代码片段来尝试它。

RDP 使用多个相互递归的函数调用，它实现了给定语法的树形结构。RDP 解析器的代码看起来像你正在处理的实际语法，只要遵循一些规则，它们就很容易编写。RDP 解析器的两个缺点是：它们可能不是非常有效，并且通常需要手动编写它们，因此它们的错误比生成的解析器更多。对于 RDP 解析器可以解析的东西，还有一些理论上的限制，但是由于你手动编写它们，你通常可以解决很多限制。

为了编写一个 RDP 解析器，你需要使用三个主要操作，来处理扫描器的记号：

> `peek`
>
> 如果下一个记号能够匹配，返回它，但是不从流中移除。
>
> `match`
>
> 匹配下一个记号，并且从流中移除。
>
> `skip`
>
> 由于不需要下个记号，跳过它，将其从流中移除。

你会注意到，这些是我在练习 33 中让你为扫描器创建的三个操作，这就是为什么。你需要他们来实现一个 RDP 解析器。

你可以使用这三个函数来编写语法解析函数，从扫描器中获取记号。这个练习的一个简短的例子是，解析这个简单的函数：

```
def function_definition(tokens):
    skip(tokens) # discard def
    name = match(tokens, 'NAME')
    match(tokens, 'LPAREN')
    params = parameters(tokens)
    match(tokens, 'RPAREN')
    match(tokens, 'COLON')
    return {'type': 'FUNCDEF', 'name': name, 'params': params}
```

你可以看到我只是接受记号并使用 `match` 和 `skip` 处理它们。你还会注意到我有一个 `parameters` 函数，它是“递归下降解析器”的“递归”部分。当它需要为函数解析参数时，`function_definition` 会调用 `parameters` 。

BNF 语法

尝试从头开始编写一个 RDP 解析器是没有某种形式的语法规范的，有点棘手。你还记得当我要求你将单个正则表达式转换成 FSM 吗？这很难吗？它需要更多的代码，不只是正则表达式中的几个字符。当你为这个练习编写 RDP 解析器时，你将会做类似的事情，因此它有助于使用一种语言，它是“语法的正则表达式”。

最常见的“语法的正则表达式”被称为 Backus–Naur Form（BNF），以创作者 John Backus 和 Peter Naur 命名。BNF 描述了所需的记号，以及这些记号如何重复来形成语言的语法。BNF 还使用与正则表达式相同的符号，所以 `*` ， `+` 和 `?` 有相似的含义。

对于这个练习，我将使用 https://tools.ietf.org/html/rfc5234 上面的 IETF 增强 BNF 语法，来规定上面的微型 Python 代码段的语法。ABNF 运算符大部分与正则表达式相同，只是由于某种奇怪的原因，它们在要重复的东西之前放置重复符号。除此之外，请阅读规范，并尝试弄清楚下面的意思：

```
root = *(funccal / funcdef)
funcdef = DEF name LPAREN params RPAREN COLON body
funccall = name LPAREN params RPAREN
params = expression *(COMMA expression)
expression = name / plus / integer
plus = expression PLUS expression
PLUS = "+"
LPAREN = "("
RPAREN = ")"
COLON = ":"
COMMA = ","
DEF = "def"
```

让我们仅仅查看 `funcdef` 那一行，并将其与 `function_definition` Python 代码比较，匹配每一个部分：

`funcdef =`

我们使用 `def function_definition(tokens)` 来复制，并且它是我们的语法的这个部分的开始。

`DEF`

它在语法中规定了 `DEF = "def"`，并且在 Python 代码中，我们使用 `skip(tokens)` 跳过了它。

`name`

我需要它，所以我使用 `name = match(tokens, 'NAME')` 匹配它。我使用 CAPITALS 的约定，在 BNF 中表示我会跳过的东西。

`LPAREN`

我假设我收到了一个 `def`，但是现在我打算确保有一个 `(`，所以我要匹配它。但是我使用 `match(tokens, 'LPAREN')` 来忽略结果。它就像“需要但是忽略”。

`params`

在 BNF 中我将 `params` 定义为了新的“语法产生式”，或者“语法规则”。意思是在我的 Python 代码中，我需要一个新的函数。这个函数中，我可以使用 `params = parameters(tokens)` 来调用那个函数。之后我定义了 `parameters` 函数来为函数处理逗号分隔的参数。

`RPAREN`

同样我需要但是去掉了它，使用 `match(tokens, 'RPAREN')`。

`COLON`

同样，我去掉了匹配 `match(tokens, 'COLON')`。

`body`

我这里实际上跳过了函数体，因为 Python 的缩进语法对于这个例子太难了。你不需要在练习中处理这个例子，除非你喜欢它。

这基本上是，你如何读取 ABNF 规范，并将其系统地转换为代码。你从根开始，将每个语法产生式实现为一个函数，并让扫描器处理简单的记号（我用 `CAPITAL`（大写）字母表示）。

简单的示例黑魔法解析器

这是我快速 Hack 出来的 RDP 解析器，你可以使用它，作为你的更正式和简洁的解析器的基础。

```
from scanner import *
from pprint import pprint

def root(tokens):
    """root = *(funccal / funcdef)"""
    first = peek(tokens)

    if first == 'DEF':
        return function_definition(tokens)
    elif first == 'NAME':
        name = match(tokens, 'NAME')
        second = peek(tokens)

        if second == 'LPAREN':
            return function_call(tokens, name)
        else:
            assert False, "Not a FUNCDEF or FUNCCALL"

def function_definition(tokens):
    """
    funcdef = DEF name LPAREN params RPAREN COLON body
    I ignore body for this example 'cause that's hard.
    I mean, so you can learn how to do it.
    """
    skip(tokens) # discard def
    name = match(tokens, 'NAME')
    match(tokens, 'LPAREN')
    params = parameters(tokens)
    match(tokens, 'RPAREN')
    match(tokens, 'COLON')
    return {'type': 'FUNCDEF', 'name': name, 'params': params}

def parameters(tokens):
    """params = expression *(COMMA expression)"""
    params = []
    start = peek(tokens)
    while start != 'RPAREN':
        params.append(expression(tokens))
        start = peek(tokens)
        if start != 'RPAREN':
            assert match(tokens, 'COMMA')
    return params

def function_call(tokens, name):
    """funccall = name LPAREN params RPAREN"""
    match(tokens, 'LPAREN')
    params = parameters(tokens)
    match(tokens, 'RPAREN')
    return {'type': 'FUNCCALL', 'name': name, 'params': params}

def expression(tokens):
    """expression = name / plus / integer"""
    start = peek(tokens)

    if start == 'NAME':
        name = match(tokens, 'NAME')
        if peek(tokens) == 'PLUS':
            return plus(tokens, name)
        else:
            return name
    elif start == 'INTEGER':
        number = match(tokens, 'INTEGER')
        if peek(tokens) == 'PLUS':
```

```
            return plus(tokens, number)
        else:
            return number
    else:
        assert False, "Syntax error %r" % start

def plus(tokens, left):
    """plus = expression PLUS expression"""
    match(tokens, 'PLUS')
    right = expression(tokens)
    return {'type': 'PLUS', 'left': left, 'right': right}


def main(tokens):
    results = []
    while tokens:
        results.append(root(tokens))
    return results

parsed = main(scan(code))
pprint(parsed)
```

你会注意到，我正在使用我写的 `scanner` 模块，拥有我的 `match`，`peek`，`skip` 和 `scan` 函数。我使用 `from scanner import *`，仅使这个例子更容易理解。你应该使用你的 `Scanner` 类。

你会注意到，我把这个小解析器的 ABNF 放在每个函数的文档注释中。这有助于我编写每个解析器代码，稍后可以用于错误报告。在尝试挑战练习之前，你应该研究此解析器，甚至可能作为“代码大师副本”。

挑战练习

你的下一个挑战是，将你的 `Scanner` 类与新编写的 `Parser` 类组合在一起，你可以派生并重新实现使我这里的简单的解析器。你的基础 `Parser` 类应该能够：

- 接受一个 `Scanner` 对象并执行自身。你可以假设任何默认函数是语法的起始。
- 拥有错误处理代码，比我简单的 `assert` 用法更好。

你应该实现 `PunyPythonPython`，它可以解析这个微型 Python 语言，并执行以下操作：

- 不是仅仅产生 `dicts` 的列表，你应该为每个语法生产式的结果创建类。这些类之后成为列表中的对象。
- 这些类只需要存储被解析的记号，但是要准备做更多事情。
- 你只需要解析这个微型语言，但你应该尝试解决“Python 缩进”问题。你可能需要秀阿贵扫描器，使其更智能，才能在行的开头匹配 `INDENT` 空白字符，并在其他位置忽略它。你还需要跟踪如何多少缩进了多少，同时也记录零缩进，所以你可以“压缩”代码块。

一个泛用的测试套件涉及到，将这个微型 python 的更多样本交给解析器，但现在只需要得到一个小文件来解析。尝试在测试中获得良好的覆盖率，并尽可能多地发现错误。

研究性学习

这个练习相当庞大，所以只需要完成。花点时间，一次做一点点。我强烈建议学习我这里的小型样本，直到你完全弄清楚，并打印正在处理的关键位置的记号。

深入学习

查看 David Beazley 的 SLY 解析器生成器，以便让你的计算机为你生成你的解析器和扫描器（也称为分词器）。随意尝试用 SLY 重复此练习来进行比较。

练习 34：分析器

> 原文：Exercise 34: Analyzers
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你现在有了一个解析器，它应该生成一个语法产生式对象树。我会将其称为“解析树”，这意味着你可以从“解析树的顶部开始，然后“遍历”它，直到你访问每个节点来分析整个程序。当你了解 `BSTree` 和 `TSTree` 数据结构时，你已经做了这样的事情。你从顶部开始访问了每个节点，并且你访问的顺序（深度优先，广度优先，顺序遍历等）确定了节点的处理方式。你的解析树具有相同的功能，编写微型 Python 解释器的下一步是遍历树并分析它。

分析器的任务是，在你的语法中找到语义错误，并修复或添加下一阶段需要的信息。语义错误是错误，虽然语法正确，但并不适合 Python 程序。这可以是一个尚未定义的遍历，也可以是不符合逻辑的代码，它根本没有意义。一些语言语法是如此松散，分析器必须做更多的工作来修复解析树。其他语言很容易解析和处理，甚至不需要分析器的步骤。

为了编写分析器，你需要一种方法来访问解析树中的每个节点，分析错误，并修复任何缺少的信息。有三种通用方法可以用于实现它：

- 你创建一个分析器，它知道如何更新每个语法产生式。它将以和解析器相似的方式遍历解析树，对每种生产式类型都拥有一个函数，但他的任务是更改，更新和检查产生式。
- 你改变你的语法产生式，让他们知道如何分析自己的状态。那么你的分析器就仅仅是一个引擎，它遍历解析树，调用每个产生式的 `analyze()` 方法。使用这种风格，你将需要一些状态，它们会传递给每个语法产生式类，这个类应该是第三个类。
- 你创建一组单独的类来实现最终分析后的树，你可以将其传递给解释器。通过许多方式，你将使用一组新的类来映射语法分析器的语法产生式，这些新的类接受全局状态，语法产生式，并配置其 `__init__`，使其为分析后的结果。

我建议你现在使用 #2 或 #3 来完成挑战练习。

访客模式

“访问者模式”是面向对象语言中非常常见的技术，其中你可以创建一些类，它们知道被“访问”时应该做什么。这可以让你将处理某个类的代码集成到这个类。这样做的优点是，你不需要大型的 `if` 语句来检查类上的类型，来了解该做什么。相反，你只需创建一个类似于这个的类：

```
class Foo(object):
    def visit(self, data):
        # do stuff to self for foo
```

一旦你拥有了这个类（ `visit` 可以叫任何东西），你可以遍历列表来调用它。

```
for action in list_of_actions:
    action.visit(data)
```

你将使用这种模式用于 #2 或 #3 风格的分析器；唯一的区别是：

- 如果你决定，你的语法产生式也将是分析结果，那么你的 `analyze()` 函数（也就是我们的 `visit()` ）只会将该数据存储在产生式类，或者在提供给它的状态中。
- 如果你决定，你的语法产生式将为解释器生成另一组类（请参阅练习 35），那么每次 `analyze` 的调用都将返回一个新对象，该对象将放入列表中以供以后使用，或将其作为子树附加到当前对象。

我将介绍第一种情况，其中你的语法产生式也是你的分析器结果。这适用于我们简单的微型 Python 脚本，你应该遵循这种风格。如果你想尝试其他的设计，那么你可以之后尝试。

简短的微型 Python 分析器

> 警告
>
> 如果你想自己尝试，为你的语法产生式尝试实现访客模式，那么你应该停在这里。我将给出一个相当完整但简单的例子，它充满了障碍。

访客模式背后的概念似乎是奇怪的，但它是完全有意义的。每个语法产生式都知道在不同阶段应该做什么，所以你可以把这个阶段代码放在需要的数据附近。为了演示这个，我写了一个小型的伪造的 `PunyPyAnalyzer` ，它仅仅使用访客模式打印出解析。我只完成一个语法产生式的样例，所以你可以理解这是如何完成的。我不想给你太多的线索。

我做的第一件事是，定义一个 `Production` 类，我的所有语法产生式都将继承它。

```
class Production(object):
    def analyze(self, world):
        """Implement your analyzer here."""
```

它拥有我的初始的 `analyze()` 方法，并接受我们随后使用的 `PunyPyWorld` 。第一个语法产生式的示例使 `FuncCall` 产生式：

```
class FuncCall(Production):

    def __init__(self, name, params):
        self.name = name
        self.params = params

    def analyze(self, world):
        print("> FuncCall: ", self.name)
        self.params.analyze(world)
```

函数调用有名称和 `params` ，它是一个 `Parameters` 产生式类，用于函数调用的参数。看看 `analyze()` 方法，你会看到第一个访客函数。当你访问 `PunyPyAnalyzer` 时，你将看到如何运行，但是请注意，此函数之后在每个函数的参数上调用 `param.analyze(world)` ：

```
class Parameters(Production):

    def __init__(self, expressions):
        self.expressions = expressions

    def analyze(self, world):
        print(">> Parameters: ")
        for expr in self.expressions:
            expr.analyze(world)
```

这就产生了 `Parameters` 类，它包含每个表达式，它们组成函数的参数。 `Parameters.analyze` 仅仅遍历它的表达式列表，其中我们拥有两个：

```
class Expr(Production): pass

class IntExpr(Expr):
    def __init__(self, integer):
        self.integer = integer

    def analyze(self, world):
        print(">>>> IntExpr: ", self.integer)

class AddExpr(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def analyze(self, world):
        print(">>> AddExpr: ")
        self.left.analyze(world)
        self.right.analyze(world)
```

在这个例子中，我只添加了两个数字，但是我创建一个基本的 `Expr` 类，然后创建 `IntExpr` 和 `AddExpr` 类。每个都仅仅拥有 `analyze()` 方法，打印出其内容。

因此，我们有用于分析树的类，我们可以做一些分析。我们需要的第一件事是一个世界，它可以跟踪变量定义、函数、以及我们的 `Production.analyze()` 方法所需的其他东西。

```
class PunyPyWorld(object):

    def __init__(self, variables):
        self.variables = variables
        self.functions = {}
```

当调用任何 `Production.analyze()` 方法时，`PunyPyWorld` 对象被传递给它，因此 `analyze()` 方法知道世界的状态。它可以更新变量，寻找函数，并在世界中执行任何所需的事情。

然后我们需要一个 `PunyPyAnalyzer`，它可以接受解析树和世界，并使所有的语法产生式运行：

```
class PunyPyAnalyzer(object):
    def __init__(self, parse_tree, world):
        self.parse_tree = parse_tree
        self.world = world

    def analyze(self):
        for node in self.parse_tree:
            node.analyze(self.world)
```

函数的简单调用 `hello(10 + 20)` 的配置相当简单。

```
variables = {}
world = PunyPyWorld(variables)
# simulate hello(10 + 20)
script = [FuncCall("hello",
            Parameters(
                [AddExpr(IntExpr(10), IntExpr(20))])
            )]
analyzer = PunyPyAnalyzer(script, world)
analyzer.analyze()
```

要确保你理解了我构造 `script` 的方式。注意到第一个参数是一个列表了嘛？

解析器与分析器

在这个例子中，我假设 `PunyPyParser` 已将 `NUMBER` 记号转换为整数。在其他语言中，你可能只拥有记号，并让 `PunyPyAnalyzer` 进行转换。这一切都取决于，你想让错误发生在哪里，以及哪里可以做最有用的分析。如果你将工作放在解析器中，那么你可以马上给出格式化方面的早期错误。如果将其放在分析器中，那么你可以给出错误，使用整个解析文件来有所帮助。

挑战练习

所有这些 `analyze()` 方法的要点不仅仅是打印出来，而是改变每个 `Production` 子类的内部状态，以便解释器可以像脚本一样运行它。你在这个练习中的任务是，接受你的语法产生式类（可能与我的不同）并进行分析。

随意借鉴我的出发点。如果需要，可以使用我的分析器和我的世界，但是你应该尝试首先编写自己的分析器。你还应该将练习 33 中的产生式类与我的比较。你的更好吗？它们能支持这种设计吗？如果他们不能则改变它们。

你的分析器需要做一些事情才能使解释器正常工作：

- 跟踪变量定义。在一个实际的语言中，这将需要一些非常复杂的嵌套表，但是对于微型 Python 来说，只需假设有一个巨大的表（ `TSTree` 或 `dict` ），所有变量都在这里。这意味着 `hello(x, y)` 函数的 `x` 和 `y` 参数实际上是全局变量。
- 跟踪函数的位置，以便以后运行它们。我们的微型 Python 只有简单的函数，但是当 `Interpreter` 运行时，它需要“跳转”到并运行它们。最好的办法保留它们，便于之后使用。
- 检查你可以想到的任何错误，例如使用中缺少的变量。这是棘手的，因为 Python 这样的语言，在解释器阶段中进行更多的错误检查。你应该决定在分析过程中，可能出现哪些错误并实现它们。例如，如果我尝试使用未定义的变量，会发生什么？
- 如果你正确地实现了 Python `INDENT` 语法，那么你的 `FuncCall` 产生式应该有额外的代码。解释器将需要它来运行它，所以确保有一个实现它的方式。

研究性学习

- 这个练习已经很难了，但是如何创建一个更好的方式，来存储变量，至少实现一个额外的作用域层级？记得“作用域”的概念是， `hello(x, y)` 中的 `x, y` 不影响 `hello` 函数之外的你定义 `x` 和 `y` 。
- 在 `Scanner` ， `Parser` 和 `Analyzer` 中实现赋值。这意味着我应该可以执行 `x = 10 + 14` ，你可以处理它。

深入学习

研究“基于表达式”和“基于语句”的编程语言之间的区别。较短版本是一些只有表达式的语言，所以任何东西都有与之相关的某种（返回）值。其他语言的表达式拥有值，语句没有，因此把它们赋给变量会失败。Python 是哪种语言？

练习 35：解释器

> 原文：Exercise 35: Interpreters
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

解析中的最后一个练习应该既具有挑战性又有趣。你终于可以看到，你的微型 Python 脚本运行并做了一些事情。难以理解这个章节和解析的概念很正常。如果你发现你已经到达了这里，而且你不太明白发生了什么，请退后一步，再考虑在这一部分做一些练习。在继续之前，重复几次这个章节，这可以帮助你在最后两个练习中制作自己的小语言。

我故意不会在本练习中包含任何代码，以便你必须根据解释器工作方式的描述来尝试。你已经有了 Python 作为参考，我们的微型 Python 示例应该如何操作这些小语句。你知道如何用访问者模式来遍历你的解析树。剩下的就是让你编写一个解释器，它可以将它们结合在一起，并使你的小脚本运行。

解释器和编译器

在编程语言的世界里，存在解释语言和编译语言。编译语言接受你的输入的源码，并进行扫描，解析和分析阶段。然后，编译器基于这个分析产生机器码，通过遍历它并编写真正的（或假的）计算机所需的字节，来使 CPU 运行。一些编译器增加了一个额外的步骤，将输入的源码翻译成通用的“中间语言”，然后将其编译为机器的字节。编译器通常是确定的，因为你通常不能仅仅运行编译器，而是首先必须通过编译器运行源代码，然后执行结果。C 是一个经典的编译器，你可以这样运行 C 程序：

```
$ cc ex1.c -o ex1
$ ./ex1
```

`cc` 命令是“C 编译器”的意思，也就是说，你选取了文件 `ex1.c`，扫描，解析，并进行分析，然后输出可执行字节到文件 `ex1`。一旦你完成了，你就可以像其他任何程序一样运行它。

解释器不会生成你运行的编译后字节码，而是直接运行并分析结果。它“解释”输入语言，就像我把我的汉语转换为我朋友的泰语。它加载源文件，然后像编译器那样进行扫描，解析和分析。之后，它只是使用解释器的自己的语言（在这里是 Python），来根据分析来运行它。

如果我们要在 Python 中实现 JavaScript 解释器，我们会“使用 Python 解释 JavaScript”。JavaScript 是我的汉语，一个解释器正在为我将其凭空解释为 Python（泰语）。如果我想用 Python 解释 JavaScript 的 `1 + 2`，我可能会这样做：

- 扫描 `1 + 2` 并产生记号 `INT(1) PLUS INT(2)` 。
- 将其解析为表达式 `AddExpr(IntExpr(1), IntExpr(2))`
- 分析它，将文本 `1` 和 `2` 转换为实际的 Python 整数。
- 使用 Python 代码 `result = 1 + 2` 解释它，我可以将其转给剩余的解析树。

与之相比，编译器会做 1~3 的任何事情。但是在第四步它会编写字节码（机器码）到另一个文件，我可以将其运行在 CPU 上。

Python 两者都是

Python 更现代化，通过几乎完成编译和解释，利用更快的计算机。它将像解释器一样工作，所以你不必经历编译阶段。但是，解释器出奇地慢，所以 Python 有一个内部的虚拟机。当你运行脚本时，例如 `python ex1.py` ，Python 实际上会运行它并将其编译到 `__pycache__` 目录中的 `ex1.cpython -36.pyc` 文件。该文件是字节码，Python 程序知道如何加载和运行，它的工作原理就像假的机器代码。

> 译者注：但是没有 JIT 的情况下还是很慢。

你的解释器永远不会，也不应该是这样的。你的解释器应该只是扫描，解析，分析和解释微型 Python 脚本。

如何编写解释器

当你编写解释器时，你将需要工作在所有三个阶段之间，来修复你错过或做错的东西。我建议你先添加数字，然后再处理更复杂的表达式，直到你的脚本能够运行。我会像这样完成它：

- 将你的第一个 `interpret` 方法添加到 `AddExpr` 类，并让它打印出一条消息。
- 让你的解释器能够可靠地访问这个类，并传入它需要的 `PunyPyWorld` 。
- 一旦你完成了，你可以调用 `AddExpr.interpret` 来计算它的两个表达式的和，并返回结果。
- 之后，你必须弄清楚，这个 `interpret` 步骤的结果应该到哪里去。为了保持简单，让我们假设微型 Python 是一种基于表达式的语言，所以一切都返回一个值。在这种情况下，对一个解释器的调用总是具有返回值，父调用可以使用它。
- 最后，由于微型 Python 基于表达式，你可以让你的 `Interpreter` 打印出其 `interpret` 调用的最终结果。
- 如果你这样做，你将会获得解释器的基础知识，你可以开始执行所有其他的 `interpret` 方法，使其运行。

挑战练习

编写微型 Python 的解释器，应该只涉及编写另一个访问者模式，它遍历分析后的解析树，并完成解析树让它做的任何事情。你唯一的目标就是让这个很小的（甚至是微型的）脚本运行。这似乎是愚蠢的，因为这只是三行代码，但它涵盖编程语言中的各种主题：变量，加法，表达式，函数定义和函数调用。如果你实现了 `if` 语句，你几乎可以有一个可工作的编程语言。

你的任务是编写一个 `PunyPyInterpreter` 类，它接受 `PunyPyWorld` 和 `PunyPyAnalyzer` 的运行结果来执行脚本。你必须实现 `print`，仅仅用于打印其变量，但其余的代码，应该在你遍历每个产生式类的时候运行。

研究性学习

- 一旦你拥有了 `PunyPyInterpreter`，你应该实现 `if` 语句和布尔表达式，然后扩展你的语言测试集，来确保这是有效的。尽可能为这个小型 Python 解释器增加功能。
- 如何使微型 Python 也拥有语句？

深入学习

你应该能够学习尽可能多的语言的语法和规范。继续寻找一些语言并学习它们，但是使用该语言的源代码来完成。你还应该完整学习 https://tools.ietf.org/html/rfc5234 上面的 IETF ABNF 规范，来为自己准备接下来的两项练习。

练习 36：简单的计算器

> 原文：Exercise 36: Simple Calculator
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

这个挑战是创建一个简单的代数计算器，使用你所学到的关于解析的一切。你将需要设计一种语言，用于使用变量进行基本数学运算，为该语言创建 ABNF，并为其编写扫描器，解析器，分析器和解释器。这实际上对于简单的计算器语言可能是小题大做，因为不会有任何嵌套的结构，如函数，但是无论如何都要理解完整的过程。

挑战练习

简单的代数语言对于不同的人来说意思也不同，所以我希望你试试 Unix 命令 `bc`。这是我运行 `bc` 命令的一个例子：

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
x = 10
y = 11
j = x * y
j
110
```

你需要创建变量，输入数字（包括整数和浮点数），并拥有尽可能多的，你可以想到的运算符。你最有可能使用 `bc`，甚至是 Python 的 shell，并且在你弄明白时候为它编写 ABNF。请记住，你的 ABNF 几乎是伪代码，不必形式上正确，只需足够接近来创建扫描器和解析器。

一旦“简单制作”了 ABNF 形式的语法，你可以坐下来创建扫描器和解析器。我会写一套简单的脚本，来练习你认为语言应该做的事情，然后让你的测试套件，在每个阶段通过你的计算器运行它们。这样做可以更容易地测试计算器。

完成解析器之后，你应该编写一个分析器来巩固吗，并检查输入的语义。在这样一种简单的语言中，它可能不仅仅是你需要的东西，但这是一个练习，用小型玩具语言完成整个过程。请记住，分析器的重要任务是，跟踪脚本中不同位置的变量定义，以便在执行过程中它们可由解释器访问。

在分析器创建可执行解析树之后，你可以编写一个运行它的解释器。如练习 35 所述，你可以使用两种方式来编写解释器。一个是你创建一个“机器”，知道如何运行语法产生式，作为一系列的输入。这将把你的语法产生式类（`Expression`，`Assignment` 等）视为机器代码，并且简单地执行它们所包含的内容。例如 Python 这样的 OOP 语言的另一种风格是，让每个产生式类知道如何运行自身。在这种风格中，这些类很“聪明”，并且接受他们的环境，只需要做他们需要做的事情，来使事情发生。然后，你只需“遍历”语法产生式列表，并调用 `run`，直到调用完毕。

你选择哪一个，决定了你在哪里存储你的小型解释器的状态。如果你制作 `Interpreter` 类，仅仅执行产生式数据对象，那么解释器可以跟踪所有的状态，但语言更难扩展，因为你必须为每个产生式类改进 `Interpreter`。如果你的产生式类知道如何执行自己的代码，那么扩展语言很容易，但是你必须找到一种方法，在每个产生式之间传递计算机状态。

处理它的时候，我建议你仅仅以一个非常小的表达式来起步，比如加法。让整个系统首先能够工作，从扫描器一直到运行简单的加法。然后，如果你不喜欢这个设计，你可以把它丢掉，使用不同的设计重做。一旦你的设计能够工作，你就可以使用更多功能来扩展语言。

研究性学习

- 最好的研究性学习是创建函数来执行计算和返回结果。如果你可以这样做，那么你的设计将可能适用于更大的语言。
- 接下来要尝试的是，使用 `if` 语句和 `boolean` 检查来实现控制流。如果这太难了，那就对了，但请试试看。

深入学习

尽你所能来研究 `bc` 或 Python 语言。尝试找到其他语法文件来阅读和学习，特别是任何 IETF 协议的描述。IETF 的规范（像湿巾那样）让人兴奋，但它们是个很好的练习。

练习 37：小型 BASIC

> 原文：Exercise 37: Little BASIC
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你现在要及时回到我的童年，并实现一个 BASIC 解释器。不，我这里的 BASIC 不是指“一个非常简单的基本的解释器”。我的意思是 BASIC 编程语言。它是最早的编程语言之一，最初由 John Kemeny 和 Thomas Kurtz 在 Dartmouth 创建。这个基本版本叫做 Dartmouth BASIC，在 Dartmouth BASIC 维基百科页面上，代码看起来像这样：

```
5 LET S = 0
10 MAT INPUT V
20 LET N = NUM
30 IF N = 0 THEN 99
40 FOR I = 1 TO N
45 LET S = S + V(I)
50 NEXT I
60 PRINT S/N
70 GO TO 5
99 END
```

左边的数字实际上是手动输入的行号。你告诉 BASIC 每行一个数字，然后你可以只是告诉它“跳到”那一行来循环。后来在其他版本的 BASIC 中成为了 `GOTO`，成为计算时代的象征。

BASIC 的最新版本，在 BASIC 维基百科页面上记载，该页面记载了这个语言的漫长演化过程，朝着越来越现代的形式。过了一段时间，它吸收了 C 和 Algol 这样的结构，然后它面向对象，今天你可以找到相当先进的 BASIC 版本。如果你想要现代的免费 BASIC，请查看 Gambas BASIC，网址为 http://gambas.sourceforge.net/en/main.html。

挑战练习

你的挑战是实现原始的 BASIC 解释器 - 具有手动行号和所有 CAPS（大写）文本样式的解释器。你需要查看 BASIC 维基百科页面，来获得可能的记号和示例代码，并阅读 Dartmouth BASIC 维基百科页面来了解更多线索。你的解释器应该能处理尽可能多的原始 BASIC 并产生有效的输出。

当你尝试这样做时，我建议你尝试简单的数学运算，打印和跟踪行号。之后，我会努力使 `GOTO` 正常工作。如果你完成它的话，你可以完成剩余部分，慢慢开发一套测试程序，来确保你的解释器工作顺利。

祝你好运！这可能花费你一段时间，但它应该很有趣。我可以看到自己花了几个月的时间在这上面，添加愚蠢的功能，像图形，所以我可以创建所有这些愚蠢的小程序，当我还是孩子的时候我编写了它们。我写了这么多 BASIC 代码，计算行号绝对扭曲了我的大脑。这可能是我这么喜欢 Vim 的原因。

研究性学习

这个练习很困难，但如果你想要一些额外的挑战，请执行以下操作：

- 使用像 SLY 这样的解析器生成器，创建一个替代的解释器。一旦你有了 ABNF，这可能会变得更加容易，但是对于 BASIC 这样的语言可能更难。你必须这样做才能弄清楚。
- 尝试制作一个“结构化 BASIC”的版本，它拥有函数，循环，`if` 语句，以及你可以在较旧的非 OOP 语言（如 C 或 Pascal）中找到的所有内容。这是一个巨大的任务，所以建议你尝试不要手写 RDP 解析器。使用像 SLY 这样的工具生成你的解析器，并为更重要的东西节省你的脑力。

第六部分：SQL 和对象关系映射

> 原文：Part VI: SQL and Object Relational Mapping
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在本书的这一部分中，我们将介绍一些内容，它们与本书其余部分的结构不相符，但对于初级开发人员来说，这是非常必要的主题。了解如何在 SQL 数据库中构造数据，会教给你如何在逻辑上思考数据存储需求。有一个建立已久的方法来解构数据，有效存储数据和访问数据。近年来 NoSQL 数据库的发展使其不同，但关系数据库设计背后的基本概念仍然有用。在你需要存储数据的每个地方，都需要良好地构造并理解数据。

大多数这些练习会让你涉及使用 SQL 数据库，因此，我建议你从 SQLite3 下载页面下载 `sqlite3` 二进制文件，如果你还没有安装的话。我们使用 Python，所以它已经安装在大多数 Python 发行版中，但有时它不可用。如果你不能在你的 python shell 中运行这个 Python 代码：

```
>>> import sqlite3
```

你的 Python 就没有默认带有 `sqlite3`。你需要弄清楚为什么会丢失，并且很可能有另外一个包，你需要先安装它才能在 Python 中使用。

理解 SQL 就是理解表

当你开始这部分的练习之前，你需要完整理解一个概念，它为许多 SQL 初学者造成了问题。

> SQL 数据库中的每个单独的东西都是一张表。

把它刻录到你的大脑里。对于“表”，我是说就像一个电子表格，其中左边有行，顶部有列。通常，你将使用进入该列的某种数据来命名列。那么每一行代表你需要放入表的一件事情。这可以是一个帐户，一个人的名单及其信息，菜谱，甚至汽车。每一行都是一辆汽车，每列是一些属性，关于你需要跟踪的那辆车。

这为大多数程序员造成了问题，因为我们按照树形结构思考问题。一个对象其中有另一个对象，对象里面有个列表，列表里面有个字典，字典里面有个字符串，字符串映射为数据。我们将东西嵌套在里面，而且这种风格的数据结构不适合表。对于大多数程序员来说，似乎这两个结构（表和树）不能共存，但是树和表实际上是非常相似的。你几乎可以使用任何树形结构，并将其映射到几乎任何矩阵上，但你必须了解 SQL 数据库的另一个方面：关系。

关系使得 SQL 数据库变得比电子表格更有用。电子表格可以让你创建一整套工作表，并在其中放置不同类型的数据，但是难以将这些工作表链接在一起。SQL 数据库的目的完全是，使你可以使用列或其他表将表链接在一起。SQL 数据库的天赋是，使用一个结构（表）来构建几乎任何类型的数据结构，你可以通过将它们链接在一起来实现。

我们将了解SQL数据库中的关系，但快速回答是，如果你可以创建一个数据树，那么你可以将该树放入1个或多个表中。在本书的这个阶段，我们可以简化将一组相关的Python类转换为SQL表的过程，如下所示：

- 为所有类创建表。
- 在子表中设置 `id` 列指向父表。
- 在任何两个类“之间”创建链接表，这两个类通过列表链接。

它比这更复杂，但是，当将一组类转换为 SQL 时，这是所做事情的要点。事实上，大部分像 Django 这样的系统，是上述三件事情的复杂版本。

你会学到什么

本节的目的不是教你如何成为一个 SQL 系统管理员。如果你想做这个工作，那么我建议你学习有关 Unix 的一切，然后去获得一个公司的证书，这个公司提供技术认证。请记住，这不是一个非常有趣的工作，类似于看管一个大型的猫类动物园。猫，不是小猫。

在第六部分末尾，你将学到 SQL 在基本层面上的工作原理。这是一个 SQL 速成课，以你创建的对象关系映射器（ORM）结束，它与 Django 相似。本节仅仅是了解 SQL 工作方式的一个突破点，目的是为你提供足够的信息，来了解 Django 系统中发生的东西。

如果你想在你的工作中超出这个部分，我推荐 Joe Celko 的《SQL For Smarties》，和一些时间。Joe 的书很厚，但很完整，他是 SQL 的大师。阅读这本书将使你非常能干。

练习 38：SQL 简介

> 原文：Exercise 38: Introduction To SQL
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

学习如何建模和设计实体数据的最佳方法，是从非常基本的搭积木开始。数据库的 SQL（“SeQueL”）风格数十年来已成为数据建模和存储的标准。一旦你知道基本的 SQL，你可以轻松地使用任何 NoSQL 或对象关系映射（ORM）系统。SQL 是一种非常形式化的存储，操作和访问数据的方式，向你提供了一种思考它的形式化方式。这也不是很困难，因为这个语言并不像完整的编程语言那样图灵完备。

SQL 无处不在，我不是因为我想让你使用它而这么说。这只是一个事实。我敢打赌，现在你的口袋里有一些 SQL。所有 Android 手机和 iPhone 都可以轻松访问名为 SQLite 的 SQL 数据库，手机上的许多应用程序都可以直接使用它。它撑起了银行，医院，大学，政府，小企业和大型企业；这个星球上的每个计算机，和每一个人最终都会接触一些运行 SQL 的东西。SQL 是一个非常成功和健壮的技术。

SQL 的问题是，每个人似乎都讨厌它的本质。大多数程序员不能忍受，这是一种奇怪的笨拙的“非语言”。在任何现代问题很久之前，比如“网络规模”或面向对象编程，他就被设计了出来。尽管基于坚实的数学构建的操作理论之上，但是它有令人讨厌的足够的错误。树？嵌套对象和父子关系？SQL只是嘲笑你，给你一个大型的扁平的表，说“你弄清楚它吧，兄弟”。

如果每个人都如此讨厌它，为什么要学习 SQL？因为这个假设的仇恨背后，是缺乏对 SQL 的理解以及如何使用它。部分 NoSQL 运动是对过时数据库服务器的反应，也是对 SQL 的恐惧的反应，它来源于对其工作原理的忽视。通过学习 SQL，你实际上将学习一些重要理论概念，它们适用于过去和现在几乎所有数据存储系统。

无论 SQL 仇恨者声称什么，你应该学习 SQL，因为它是无处不在的，实际上并不足够难以学习。成为博学的 SQL 用户，将帮助你为要使用的数据库做出明智的决定，无论是否使用 SQL，并且作为程序员，更深入地了解你使用的许多系统。

SQL 是什么？

我将 SQL 读作“Sequal”，但如果你愿意也可以读作“S-Q-L”。SQL 也代表结构化查询语言，但现在还没有人甚至关心，因为那只是一个营销手段。SQL 所做的事情，只是为你提供了一种语言，用于与数据库中的数据交互。然而，它的优势在于，它匹配了许多年前建立的理论，定义了良好结构化数据的属性。这不完全相同（一些诋毁者感叹它），但它足够有用。

> 译者注：不要理会那些让你读成“S-Q-L”的人，就算标准是这样，你可以把“Sequal”当做别名。

SQL 的工作原理是，它了解表中的字段，以及如何根据字段的内容在表中查找数据。所有 SQL 操作都是你对表执行的四个常规操作之一：

名称	中文缩写	首字母	意义
创建	增	C	将数据放入表中
读取	查	R	从表中查询数据
更新	改	U	修改已经在表中的数据
删除	删	D	从表中移除数据

这缩写为“CRUD”，被认为是每个数据存储系统必须具备的基本功能。事实上，如果你不能以某种方式来执行这四种之一，那么最好有一个很好的理由。

> 译者注：一些人把它们简写为 CURD 或者 CRUD，其实都是一样的。

我喜欢通过将其与 Excel 等电子表格软件进行比较，来解释 SQL 的工作原理：

- 数据库是整个电子表格文件。
- 表格是电子表格中的标签/表格，每个表格都有一个名称。
- 列就是列。
- 行就是行。
- 然后，SQL为你提供了一种语言，用于对其进行 CRUD 操作，来生成新表或更改现有表。

最后一条是重要的，不了解这个会使人们产生问题。SQL 只知道表，每个操作都生成表。它通过修改现有表来“生成”表，或者返回一个新的临时表作为数据集。

在阅读本书时，你将开始了解此设计的意义。例如，面向对象语言与 SQL 数据库不匹配的原因之一是，OOP 语言围绕图来组织，但 SQL 只希望返回表。虽然可以将几乎任何图形映射到表格，反之亦然，但它为 OOP 语言增加了翻译负担。如果 SQL 返回一个嵌套数据结构，那么这不会是一个问题。

起步

我们将使用 SQLite3 作为本节的练习工具。SQLite3 是一个完整的数据库系统，具有几乎无需设置的优点。你只需下载一个二进制文件，就像大多数其他脚本语言一样使用它。有了它，你将能够学习 SQL，而不会卡在数据库服务器的管理。

安装 SQLite3 很简单：

- 请访问 SQLite3 下载页面，并为你的平台获取二进制文件。寻找“Precompiled Binaries

for X”，X 是你的操作系统的首选项。
- 或使用你的操作系统的软件包管理器进行安装。如果你使用 Linux ，那么你知道这是什么意思。如果你使用 macOS ，那么首先得到一个包管理器，然后使用它来安装 SQLite3。

安装完成后，请确保你可以启动命令行并运行它。这是一个快速测试，你可以尝试：

```
$ sqlite3 test.db
SQLite version 3.7.8 2011-09-19 14:49:19
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table test (id);
sqlite> .quit
```

然后看到 `test.db` 文件在那里。如果它可以工作，那么你就完成了。你应该确保你的 SQLite3 版本与我在这里的版本相同：3.7.8。有时，旧版本的东西不能正常工作。

学习 SQL 词汇

要开始学习SQL，你需要为这些 SQL 术语创建速记卡（或使用 Anki）。在这之后的练习中，你将学习这些 SQL 语句，并将其应用于不同的问题。思考 SQL 语言的最佳方法是，将所有东西看做 `CREATE` ，`READ` ，`UPDATE` 和 `DELETE` 操作。即使一个单词是 `INSERT` ，你仍然会将其视为 `CREATE` 操作，因为它将创建数据。首先，只要花一些时间记住这些单词，并继续研究，就像本节的练习一样。

> `CREATE`
>
> 创建数据库的表格，可以储存数据的列。
>
> `INSERT`
>
> 向数据库表格添加行，并填充在数据的列中。
>
> `UPDATE`
>
> 修改表中的一列或者多列。
>
> `DELETE`
>
> 从表中删除一行。
>
> `SELECT`
>
> 查询一个表或一系列表，返回带有结果的临时表。
>
> `DROP`
>
> 销毁一个表。
>
> `FROM`
>
> SQL 语句的常见部分，用于指定要使用表的那些列。
>
> `IN`
>
> 用于表示元素集合。
>
> `WHERE`
>
> 用在查询中，来表示一些东西应该来自哪里。
>
> `SET`
>
> 用在更新中，来表示哪一列修改成什么。

SQL 语法

接下来，你将为 SQL 的另一组重要语法结构创建速记卡。他们不会太多，但是写下它们（或使用 Anki），并开始研究他们，以便你更快地学习语言。你正在学习的语法用于 SQLite3，我们将在本书中使用它。这是一个相当普遍的 SQL 语法，但每个数据库都有不同的奇怪的偏好，你必须学习它。一旦了解它，很容易弄清楚另外一个数据库的用法。

你将需要访问 SQLite 3 定义页面来创建所需的卡。该页面列出了 SQLite 了解的所有内容，但仅关注上面列出的主要语句。添加你不明白的其他任何单词。他们的图表有点复杂，但它们只是 SQL BNF 的图形视图，你在第五部分中了解了它们。如果你不记得 ABNF，返回第五部分并重新学习。

深入学习

- 访问 SQLite3 语法列表并浏览所有可用的命令。他们中的大多数都不会有意义，但是如果你有任何兴趣，那么你也可以为他们做速记卡。
- 在完成剩余练习的整个时间里，研究这些速记卡。

练习 39：SQL 创建

> 原文：Exercise 39: Creating with SQL
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

当我们谈论首字母缩写“CRUD”时，“C”代表“创建”，它不仅仅意味着创建表。这也意味着将数据插入到表中，并使用表和插入来链接表。由于我们需要一些表和一些数据来完成其余的 CRUD（增删改查），我们开始学习如何在 SQL 中执行最基本的创建操作。

表的创建

我在简介中说，可以对表内的数据执行“增删改查”操作。你如何把表放在首要位置？通过对数据库纲要（Schema）执行 CRUD ，第一个要学习的 SQL 语句是 `CREATE` ：

```
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);
```

你可以将其放在一行中，但是我打算讨论每一行，所以写成了多行。这里是每行所做的事情：

`ex1.sql:1`

`CREATE TABLE` 的起始，它提供了表的名称 `person` 。这个部分之后，之后将你想要的字段放到括号里。

`ex1.sql:2`

`id` 列，它用于准确确定每一行。列的格式是 `NAME TYPE` ，并且这里我假设，我需要一个 `INTEGER` 也是 `PRIMARY KEY` 。这样做告诉 SQLite3 来将其特殊对待。

`ex1.sql:3~4`

`first_name` 和 `last_name` 列。它们都是 `TEXT` 。

`ex1.sql:5`

`age` 列，只是一个 `INTEGER` 。

`ex1.sql:6`

使用圆括号结束列的列表，之后是一个分号（`;`）。

创建多表的数据库

创建一个表不是特别实用。我希望你现在创建三个表，你可以在里面储存数据。

```
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);

CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER
);

CREATE TABLE person_pet (
    person_id INTEGER,
    pet_id INTEGER
);
```

在此文件中，你正在为两种数据类型制作表，然后将它们与第三个表“链接”在一起。人们称这些“链接”表为“关系”，但没有生命的非常愚蠢的人把所有表都成为“关系”，并且热衷于使那些想要完成工作的人困惑。在我的书中，具有数据的表是“表”，将表连接在一起的表称为“关系”。

这里没有任何新东西，除非你看到`person_pet`，你会看到我已经写了两列：`person_id`和`pet_id`。将两个表链接在一起，只是向`person_pet`插入一行。它拥有两行的 ID 列的值，你想要链接它们。例如，如果`person`包含一行`id=20`，`pet`有一行`id=98`，然后假设这个人拥有这个宠物，你会将`person_id=20, pet_id=98`插入到`person_pet`关系（表）中。

在接下来的几个练习中，我们将实际插入这样的数据。

插入数据

你有了要处理的几个表，所以现在我让你使用`INSERT`命令，放进去一些数据：

```
INSERT INTO person (id, first_name, last_name, age)
    VALUES (0, "Zed", "Shaw", 37);

INSERT INTO pet (id, name, breed, age, dead)
    VALUES (0, "Fluffy", "Unicorn", 1000, 0);

INSERT INTO pet VALUES (1, "Gigantor", "Robot", 1, 1);
```

在这个文件中，我使用两种不同形式的 `INSERT` 命令。第一种形式是更明确的风格，最有可能是你应该使用的东西。它指定要插入的列，后跟 `VALUES`，然后要包括的数据。这两个列表（列名和值）都在括号内，并以逗号分隔。

第七行的第二个版本是一个缩写版本，它不指定列，而是依赖于表中的隐式顺序。这种形式是危险的，因为你不知道你的语句实际访问哪一列，并且某些数据库对列没有可靠的排序。当你真的很懒惰时，最好只用这种形式。

插入引用数据

在最后一节，你会在表中放满人和宠物。唯一缺少的东西是，谁拥有什么宠物，这个数据存入 `person_pet` 表，如下所示：

```
INSERT INTO person_pet (person_id, pet_id) VALUES (0, 0);
INSERT INTO person_pet VALUES (0, 1);
```

我再次使用显式格式，然后使用隐式格式。我使用我想要的 `person` 表的行 `id`（这里是 `0`），和我想要的 `pet` 表的行 `id`（同样，`0` 是独角兽，`1` 是死去的机器人）。然后，我们向 `person_pet` 关系表中插入一行，用于人与宠物之间的每个“连接”。

挑战练习

- 创建另一个数据库，但为其它东西创建其他 `INTEGER` 和 `TEXT` 字段，`person` 可能拥有它们。
- 在这些表中，我创建了第三个关系表来链接它们。你如何摆脱这个关系表 `person_pet`，并将这些信息优雅放在 `person` 里面？这个变化暗示了什么？
- 如果你可以把一行放入 `person_pet`，你是否可以放多行？你如何记录一个疯狂的猫女士与 50 只猫？
- 为人们可能拥有的汽车创建另一个表，并创建其对应的关系表。
- 在你喜欢的搜索引擎中搜索“sqlite3 数据类型”，然后阅读 SQLite3 文档中的数据类型。记录你可以使用什么类型，以及其他看起来很重要的东西。我们稍后会介绍。
- 插入你自己和你的宠物（或像我这样的虚拟宠物）。
- 如果将上一个练习中的数据库更改为没有 `person_pet` 表，则使用该模式创建一个新数据库，并将相同的信息插入到该数据库中。
- 回顾数据类型列表，并记录不同类型所需的格式。例如，请注意你有多少种方式来写入 `TEXT` 数据。
- 为你和你的宠物添加关系。
- 使用这张表，一只宠物可以被多于一个人拥有吗？这在逻辑上是可能的吗？家养的狗如何呢？严格来说，家庭中的每个人不是拥有它吗？
- 考虑上面的东西，并且考虑到你有一个替代设计，将 `pet_id` 放在 `pearon` 表中，哪种设

计更适合这种情况？

深入学习

请阅读 SQLite3 `CREATE` 命令的文档，然后查看尽可能多的其他 `CREATE` 语句。你还应该阅读 https://sqlite.org/lang_insert.html 上的 `INSERT` 文档，这应该会引导你阅读许多其他页面。

练习 40：SQL 读取

> 原文：Exercise 40: Reading with SQL
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在 CRUD 矩阵中，你只知道“创建”。你可以创建表，你可以在这些表中创建行。现在我将告诉你如何“读取”，或者在 SQL 中是 `SELECT`：

```sql
SELECT * FROM person;

SELECT name, age FROM pet;

SELECT name, age FROM pet WHERE dead = 0;

SELECT * FROM person WHERE first_name != "Zed";
```

这里是每一行做的事情：

`ex5.sql:1`

这表示“从 `person` 中选择所有列并返回所有行”。`SELECT` 的格式是 `SELECT what FROM tables(s) WHERE (tests)`，`WHERE` 子句是可选的。`*`（星号）字符是你想要的所有列。

`ex5.sql:3`

这里我只要从 `pet` 表请求两列，`name` 和 `age`。它将返回所有行。

`ex5.sql:5`

现在我正在从 `pet` 寻找相同的列，但是我只请求 `dead = 0` 的行。这会给我所有的活着的宠物。

`ex5.sql:7`

最后，我从 `person` 选择所有列，就像在第一行，但我现在指明，它们不等于 `"Zed"`。`WHERE` 子句决定哪一行返回，哪一行不返回。

选择多表

希望你现在专注于选择数据。永远记住这一点：SQL 只知道表。SQL 喜欢表。SQL 仅返回表。表，表，表，表！ 我以这种非常疯狂的方式重复一遍，以便你将开始意识到，你在编程中知道的东西不会有帮助。你在编程中处理图，在 SQL 中处理表。他们是相关的概念，但心智模型是不同的。

这里是一个例子，它们哪里不一样。假设你想知道 Zed 拥有什么宠物。你需要写一个 `SELECT` ，在 `person` 中查找，然后“以某种方式”找到我的宠物。为此，你必须查询 `person_pet` 表来获取所需的 `id` 列。以下是我的做事方式：

```
SELECT pet.id, pet.name, pet.age, pet.dead
    FROM pet, person_pet, person
    WHERE
    pet.id = person_pet.pet_id AND
    person_pet.person_id = person.id AND
    person.first_name = "Zed";
```

现在它看起来很庞大，但我会把它拆解，所以你可以看到，他只是简单构造新的表，基于三个表中的数据，和 `WHERE` 子句。

`ex6.sql:1`

我仅仅想要 `pet` 中的一些列，所以我在选择中指定它们。在上一个练习中，你使用 `*` 来表示“每一列”，但它在这里是一个坏主意。相反，你想要明确地指定你想要的每个表中的哪个列，你可以使用 `table.column` 实现它，就像 `pet.name` 。

`ex6.sql:2`

为了将 `pet` 连接到 `person` ，我需要遍历 `person_pet` 关系表。在 SQL 中，这意味着我需要在 `FROM` 之后列出所有三个表。

`ex6.sql:3`

`WHERE` 子句的开始。

`ex6.sql:4`

首先，我将 `pet` 连接到 `person_pet` ，通过相关 ID 列 `pet.id` 和 `person_pet.id` 。

`ex6.sql:5`

并且我需要以相同的方式，将人 `person` 连接到 `person_pet` 。现在，数据库可以仅仅搜索 id 列全部匹配的行，这些就是连接的行。

`ex6.sql:6`

我最后仅仅请求自己拥有的宠物，通过为我的名称添加 `person.first_name` 测试。

挑战练习

- 写一个查询，查找所有超过 10 年的宠物。

- 写一个查询，查找所有比你年轻的人。然后查找比你年长的人。
- 编写一个查询，`WHERE` 子句中使用多于一个测试，使用 `AND` 来编写它。例如 `WHERE first_name = "Zed" AND age > 30` 。
- 执行另一个查询，使用三个条件，并使用 `AND` 和 `OR` 运算符来搜索行。
- 如果你已经知道像 Python 或 Ruby 这样的语言，这可能是一个查看数据的令人惊奇的方式。花时间使用类和对象来构建相同的关系，然后将其映射到此配置。
- 执行一个查询，查找你到目前为止添加的宠物。
- 更改查询来使用你的 `person.id` 而不是 `person.name` ，像我一样。
- 浏览运行的输出，并确保你知道哪些 SQL 命令生成了哪个表，以及如何生成该输出。

深入学习

通过阅读 `SELECT` 命令的文档，继续深入了解 SQLite3，同时阅读 `EXPLAIN QUERY PLAN` 功能的文档。如果你不知道为什么 SQLite3 做了一些事情，`EXPLAIN` 是你的答案。

练习 41：SQL 更新

> 原文：Exercise 41: Updating with SQL
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

现在，你了解了 CRUD 的 CR 部分，还剩下更新和删除操作。与所有其他 SQL 命令一样，`UPDATE`命令遵循类似于`DELETE`的格式，但它会更改行中的列，而不是删除它们。

```
UPDATE person SET first_name = "Hilarious Guy"
    WHERE first_name = "Zed";

UPDATE pet SET name = "Fancy Pants"
    WHERE id=0;

SELECT * FROM person;
SELECT * FROM pet;
```

在上面的代码中，我将我的名字改为`"Hilarious Guy"`，因为这更准确。为了展示我的新绰号，我将我的独角兽更名为`"Fancy Pants"`。他喜欢它。

这不应该很难弄清楚，只是以防万一，我拆解第一个：

- 以`UPDATE`开始，这是你将要更新的表，这里是`person`。
- 接下来使用`SET`来说明，哪些列应该被设置为什么值。只要你用逗号分隔，如`first_name = "Zed", last_name = "Shaw"`，你可以按需更改尽可能多的列。
- 然后指定一个`WHERE`子句，为每行提供一个`SELECT`风格的测试集。当`UPDATE`找到匹配时，它执行更新，并会将列`SET`为你规定的样子。

复杂表的更新

在上一个练习中，我让你使用`UPDATE`执行子查询，现在我要求你，将所有我拥有的宠物的名称更改为`"Zed's Pet"`。

```
SELECT * FROM pet;

UPDATE pet SET name = "Zed's Pet" WHERE id IN (
    SELECT pet.id
    FROM pet, person_pet, person
    WHERE
    person.id = person_pet.person_id AND
    pet.id = person_pet.pet_id AND
    person.first_name = "Zed"
);

SELECT * FROM pet;
```

这是根据另一个表的信息更新一个表的方法。还有其他一些方法，可以做同样的事情，但是这样做是最容易理解。

更新数据

我将向你展示一种插入数据的替代方式，有助于原子地替换一行。你不一定经常需要它，但是如果必须替换整个记录，并且不希望在不使用事务的情况下执行更复杂的UPDATE，那么它将会有所帮助。

这里，我想用另一个人替换我的记录，但仅仅保留 ID。问题是我必须在事务中执行 `DELETE/INSERT` 才能使其成为原子，否则我需要执行一个完整的 `UPDATE` 。

另一个更简单的方法是使用 `REPLACE` 命令，或者将其添加到 `INSERT` 作为修饰符。这里有一些SQL，我首先无法插入新的记录，然后我使用这两种形式的 `REPLACE` 来实现它：

```
/* This should fail because 0 is already taken. */
INSERT INTO person (id, first_name, last_name, age)
    VALUES (0, 'Frank', 'Smith', 100);

/* We can force it by doing an INSERT OR REPLACE. */
INSERT OR REPLACE INTO person (id, first_name, last_name, age)
    VALUES (0, 'Frank', 'Smith', 100);

SELECT * FROM person;

/* And shorthand for that is just REPLACE. */
REPLACE INTO person (id, first_name, last_name, age)
    VALUES (0, 'Zed', 'Shaw', 37);

/* Now you can see I'm back. */
SELECT * FROM person;
```

挑战练习

- 使用 `UPDATE` ，通过我的 `person.id` ，将我的名字改回 `"Zed"` 。
- 写一个 `UPDATE` ，将任何死亡动物重命名为 `"DECEASED"` 。如果你尝试说他们是 `"DEAD"` ，它会失败，因为 SQL 会认为你的意思是，将其设置为名为 `"DEAD"` 的列，这不是你想要的。

- 尝试使用一个子查询，比如在 `DELETE` 中。
- 访问 SQL As Understood By SQLite 页面，并开始阅读 `CREATE TABLE`，`DROP TABLE`，`INSERT`，`DELETE`，`SELECT` 和 `UPDATE` 的文档。
- 尝试在这些文档中找到一些有趣的事情，并记录你不明白的事情，以便你可以稍后研究它们。

练习 42：SQL 删除

> 原文：Exercise 42: Deleting with SQL
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

这是最简单的练习，但我希望你键入代码之前思考一秒钟。如果你将 `SELECT` 写成 `"SELECT * FROM"`，将 `INSERT` 写成 `"INSERT INTO"`，那么你会怎么编写 `DELETE` 格式？你可以看下面，但是试着猜测它会是什么，然后看一看。

```sql
/* make sure there's dead pets */
SELECT name, age FROM pet WHERE dead = 1;

/* aww poor robot */
DELETE FROM pet WHERE dead = 1;

/* make sure the robot is gone */
SELECT * FROM pet;

/* let's resurrect the robot */
INSERT INTO pet VALUES (1, "Gigantor", "Robot", 1, 0);

/* the robot LIVES! */
SELECT * FROM pet;
```

我只是简单地通过删除它，然后使用 `dead=0` 将记录放回去，来为机器人实现非常复杂的更新。在以后的练习中，我将向你展示，如何使用 `UPDATE` 来实现它，所以不要以为这是更新的真正方法。

你已经熟悉了这个脚本中的大多数行，除了第五行。这里你拥有 `DELETE`，它与其他命令格式几乎相同。你提供了 `DELETE FROM table WHERE tests`，以及一种方式，将其看做移除行的 `SELECT`。任何在 `WHERE` 子句中有效的内容在这里都有效。

使用其它表来删除

记得我说过：“`DELETE` 就像 `SELECT`，但它从表中删除行。”限制是一次只能从一个表中删除。这意味着为了删除所有宠物，你需要执行一些额外的查询，然后基于它们删除。

一种方法是使用一个子查询，根据你已经编写的查询来选择要所需的 ID。还有其他的方法可以实现它，但是现在你可以根据你所知道的方法来实现它：

```
DELETE FROM pet WHERE id IN (
    SELECT pet.id
    FROM pet, person_pet, person
    WHERE
    person.id = person_pet.person_id AND
    pet.id = person_pet.pet_id AND
    person.first_name = "Zed"
);

SELECT * FROM pet;
SELECT * FROM person_pet;

DELETE FROM person_pet
    WHERE pet_id NOT IN (
        SELECT id FROM pet
    );

SELECT * FROM person_pet;
```

第 1~8 行是正常起步的 `DELETE` 命令，但是 `WHERE` 子句使用 `IN` ，匹配 `pet` 中的 `id` 列与子查询中返回的表。子查询（也称为子选择）是正常的 `SELECT` ，在尝试寻找人们拥有的宠物时，它应该看起来和以前你做的那个相似。

第 13~16 行中，然后我使用子查询，将任何不存在的宠物从 `person_pet` 表中给删除，使用 `NOT IN` 而不是 `IN` 。

SQL 处理它的方式是以下过程：

- 运行末尾处括号中的子查询，并创建一个表，带有所有列，就像普通 `SELECT` 一样。
- 将此表视为一种临时表，来匹配 `pet.id` 列。
- 浏览 `pet` 表，并删除拥有此临时表中（ `IN` ）的 ID 的任何行。

挑战练习

- 将所有 `ex2.sql` 到 `ex7.sql` 合并到一个文件中，并重执行上述脚本，以便你只需运行一个新文件即可重新创建数据库。
- 添加一些东西到脚本中，来删除其他宠物，然后再次使用新值插入它们。记住，这不是你通常更新记录的方式，只是为了练习。
- 练习编写 `SELECT` 命令，然后将它们放在 `DELETE WHERE IN` 中，来删除找到的记录。尝试删除你拥有的任何死亡宠物。
- 反着操作，删除有死亡宠物的人。
- 你真的需要删除死的宠物吗？为什么不在 `person_pet` 中移除他们的关系，并标记它们死了？写一个查询，从 `person_pet` 中去除死亡宠物。

深入学习

出于完整性，你需要阅读 `DELETE` 文档。

练习 43：SQL 管理

> 原文：Exercise 43: SQL Administration
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

“管理”一词在数据库中重载了。它的意思是“确保 PostgreSQL 服务器保持运行”，或者是“为新软件的部署更改和迁移表”。在本练习中，我只介绍如何更改和迁移简单的纲要。管理完整数据库服务器不在本书的范围之内。

销毁和更改表

您已经遇到了 `DROP TABLE` 作为删除表的方式。我将向您展示另一种使用方式，以及如何使用 `ALTER TABLE` 在表中添加或删除列。

```
/* Only drop table if it exists. */
DROP TABLE IF EXISTS person;

/* Create again to work with it. */
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);

/* Rename the table to peoples. */
ALTER TABLE person RENAME TO peoples;

/* Add a hatred column to peoples. */
ALTER TABLE peoples ADD COLUMN hatred INTEGER;

/* Rename peoples back to person. */
ALTER TABLE peoples RENAME TO person;

.schema person

/* We don't need that. */
DROP TABLE person;
```

我正在对表进行一些虚假更改，来演示这些命令，但使用 `ALTER TABLE` 和 `DROP TABLE` 语句，这是你可以在 SQLite3 中执行的一切事情。我会讲解它，让你了解发生了什么：

`ex21.sql:2`

使用 `IF EXISTS` 修饰符，仅当表已经存在时，才会丢弃。当你在没有表的新的数据库上运行你的 `.sql` 脚本时，这抑制了错误。

`ex21.sql:5`

仅仅重新创建表来处理它。

`ex21.sql:13`

使用 `ALTER TABLE` 来将其重命名为 `peoples` 。

`ex21.sql:16`

向新命名的表 `peoples` 中添加新的一列 `hatred` ，它是个 `INTEGER` 。

`ex21.sql:19`

将 `peoples` 重命名回到 `person` ，因为这对于表来说是个愚蠢的名称。

`ex21.sql:21`

转储 `person` 的纲要，所以你可以看到，它拥有新的 `hatred` 列。

`ex21.sql:24`

在这个练习之后，丢弃这个表来打扫干净。

迁移和演化数据

我们来应用您学到的一些技巧。我会让你选取你的数据库，并将纲要“演进”成不同的形式。你需要确保你很好地了解了以前的练习，并且让你的 `code.sql` 正常工作。如果你没有完成每一个这些东西，那么回去把所有东西都弄清楚。

为了确保你在正确的状态中尝试这个练习，当你运行你的 `code.sql` ，你应该可以运行 `.schema` ，像这样：

```
$ sqlite3 ex13.db < code.sql
$ sqlite3 ex13.db .schema
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);
CREATE TABLE person_pet (
    person_id INTEGER,
    pet_id INTEGER
);
CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER,
    dob DATETIME
);
```

确保你的表像我的表。并且如果不是，返回去并且移除任何命令，它们在上一个练习之后执行 `ALTER TABLE` 或者任何东西。

挑战性练习

您所要完成的任务是数据库更改的以下列表：

- 向 `person` 添加 `dead` 列，就像 `pets` 中那样。
- 向 `person` 添加 `phone_number` 列。
- 向 `person` 添加 `salary` 列，它是 `float` 。
- 向 `person` 和 `pet` 添加 `dob` 列，它是 `DATETIME` 。
- 向 `person_pet` 添加 `purchased_on` 列，它是 `DATETIME` 。
- 向 `pet` 添加 `parent` 列，它是 `INTEGER` ，并且持有它父母的 `id` 。
- 使用 `UPDATE` 语句，使用新的列数据更新现有的数据库记录。不要忘记 `person_pet` 关系表中的 `purchased_on` 列，来表明这个人什么时候购买这个宠物。
- 再增加四个人和五个宠物，并为它们分配从属关系，以及哪个宠物是父母。在最后一部分，请记住，您获取父母的 ID，然后将其设置在 `parent` 列中。
- 写一个查询，寻找 2004 年以后购买的所有宠物及其所有者的名字。关键是基于 `purchased_on` 列将 `person_pet` 映射到 `pet` 和 `parent` 。
- 写一个查询，寻找给定宠物的父母。再次查看 `pet.parent` 来实现它。实际上很简单，所以不要小题大做。
- 更新你的 `code.sql` 文件，你已经把所有的代码放了进去，让它使用 `DROP TABLE IF EXISTS` 语法。
- 使用 `ALTER TABLE` ，向 `person` 添加 `height` 和 `weight` 列，并将其放在你的 `code.sql` 文件中。
- 运行新的 `code.sql` 脚本来重置数据库，你应该没有错误。

您应该通过编写一个 `ex13.sql` 文件来实现，文件里面是这些新的东西。然后通过使用 `code.sql` 重置数据库来测试它，然后运行 `ex13.sql` 来更改数据库，并执行 `SELECT` 查询来确认您进行了正确的更改。

深入学习

继续阅读 `DROP TABLE` 和 `ALTER TABLE` 的文档，然后访问 SQLite3 语言页面，并且阅读文档的其余 `CREATE` 和 `DROP` 语句。

练习 44：使用 Python 的数据库 API

> 原文：Exercise 44: Using Python's Database API
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

Python 具有标准化的数据库 API，可以使用相同的代码访问多个数据库。你要连接的每个数据库都有一个不同的模块，它们知道如何与该数据库通信，并遵循 https://www.python.org/dev/peps/pep-0249/PEP 中的标准。这使得我们更容易使用所有数据库来访问它们，它们具有不同 API。对于本练习，你将使用 https://docs.python.org/2/library/sqlite3.html 上的 `sqlite3` 模块来处理 SQL。

学习 API

作为程序员，你必须不断做到的一件事是，学习其他人写的 API。我没有具体涵盖最有效的方式来做到它，因为大多数程序员得心应手，就像学习语言那样。Python 语言及其模块密切相关，当你学习 Python 时，你不得不学习这些模块中的 API。然而，有一种有效的方式来学习我使用的 API，在这个练习中你将要学习它。

为了学习像 `sqlite3` 模块的API，我会这样做：

- 查找 API 的所有文档，如果没有文档，请查找代码。
- 检查样例或测试代码，并将其复制到我自己的文件中。通常阅读是不够的。我实际上会使其工作，猜猜为什么，因为很多时候文档不匹配当前版本的 API。制作文档中的所有东西，可以帮助我找到所有忘记提到的内容。
- 当你获取样例代码，来工作于我的机器时，记录下任何对我有用（WFM）的情况。WFM 是，编写文档的人留下了重要的配置步骤，因为他们的计算机已经配置好了。大多数编写文档的程序员并不是从一台新机器开始，所以他们遗漏了一些库和软件，它们安装了但是别人没有。当你尝试在生产环境中配置 API 时，这些 WFM 的差异之后会阻碍你，所以我会记下来便于以后使用。
- 为所有主要 API 入口点，以及它们所做的东西制作闪存卡或笔记。
- 尝试写一个小型的峰值测试，使用 API 但只使用你的笔记。如果你点击了你不记得的 API 的一部分，请返回到文档并更新你的笔记。
- 最后，如果 API 很难使用，我会考虑使用一个简单的 API 来“包装”它，它只做我需要的东西，所以我可以忘记它。

如果这样不能学到 API，那么你应该考虑找一个不同的 API 来使用。如果 API 的作者告诉你“阅读代码”，则可能有另一个具有文档的项目。去使用该项目吧。如果你必须使用这个 API，那么考虑根据自己的代码来记录你的笔记，然后写一本书来卖，从作者的懒惰中赚钱。

挑战练习

你将以这种方式学习 `sqlite3` API，然后尝试编写自己的数据库简化 API。请记住，DB API 2.0 已经是一个不错简单的 AP，用于访问数据库I，所以你只需练习包装一个糟糕的 API。你的目标应该是充分学习 `sqlite3` API，然后设计一种更简单的方法来访问它。

有时“简单”纯粹是主观的，或是根据当前的需要。你可以决定，你需要简化的东西，不是与 SQL 数据库通信的方式，而是你与 SQL 数据库通信的方式。如果你的应用程序只需要处理人员和宠物，那么你的简化可以仅仅是，制作一个仅适用于你的 API。

深入学习

阅读 Python 中其它数据库的 API。你可以阅读 Pyscopg PostgreSQL API，以及 MySQL Python 驱动。

练习 45：创建 ORM

> 原文：Exercise 45: Creating an ORM
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

这本书的 SQL 部分的最后一个练习是一个很大的跳跃。你使用一个数据库知道了 SQL 语言的基础知识。你也应该精通 Python 的 OOP。现在是时候组合这两个，并创建一个对象关系管理器（ORM）。ORM 的工作是，使用简单的 Python 类，并将它们转换为数据库表中存储的行。如果你曾经使用过 Django，那么你已经使用他们的 ORM 来存储数据。在本练习中，你将尝试逆向分析如何实现它。

挑战练习

在现实世界中，如果一个为我工作的程序员打算创建自己的 ORM，我会说："没门，使用现有的。"工作环境不同于教育环境，因为有人付钱让你完成一些事情。使用你的工作时间，来创造一个不能使你的雇主受益的事情，这是不正当的。但是，你自己的个人时间全部是你的，作为初学者，你应该尝试重新创建尽可能多的经典软件。

创建一个 ORM 将会让你了解许多问题，关于面向对象概念和 SQL 之间的不一致。有许多 SQL 可以建模的东西，而类经常卡在这里。还有一个问题，SQL 中的一切都是表。尝试创建自己的 ORM 将会让你深入了解 SQL 和 OOP，我建议花费大量的时间，尽你所能制作一个最好的 ORM。

你在 ORM 中应实现的一些主要功能有：

- 从外部传递字符串到 ORM 应该是安全的。如果你使用 F 字符串来制作你的 SQL，那么你就错了。原因是，如果你执行 `f"SELECT * FROM {table_name}`，那么有人可以从外部将 `table_name` 设置为 SQL，例如 `person; DROP TABLE person`。你的数据库很可能以这种方式运行，销毁所有内容或更糟。有些数据库甚至允许你在 SQL 中运行系统命令，这被称为"SQL 注入"，你不应该在 ORM 中引入它。
- 所有的 CRUD 操作，但在 Python 中实现。我建议你跳过 `CREATE TABLE` 部分，直到你让其他的一切正常工作。简单的 `INSERT`，`SELECT`，`UPDATE` 和 `DELETE` 是易于制作的，但是从类定义创建数据库纲要涉及到一些主流的 Python 黑魔法，使其真正有效。使用手工制作的 `.sql` 文件创建你的数据库，然后一旦让其他东西正常工作，你可以尝试纲要系统来替换 `.sql` 文件。
- 将 Python 类型匹配到 SQL 类型以及新类型，来处理 SQL 类型。你可能会发现，你必须

做一些杂技，将 Python 数据类型放到 SQL 表中。也许这太痛苦了，所以你最终会自己制作数据类型。这就是 Django 做的事情。
- 事务是一个高级话题，但如果你可以实现它就试一试。

我也会说，在这个练习中，你可以从任意数量的项目借鉴功能。在设计时，请随意查看 Django 的 ORM。最后，我强烈建议你首先仅仅实现一个 ORM，它可以处理你在本书的这个部分创建的小型数据库。一旦你得到一个可以处理这个数据库的东西，就可以将其推广到任何数据库。

深入学习

本书开头提到了，Joe Celko 的《"Smarties" SQL》将会让你了解，你需要了解的 SQL 的每一件事情。Joe 的书很好，会让你远远超出这个小型速成课。

第七部分：大作业

> 原文：Part VII: Final Projects
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

本书的最后一部分是你进阶更高级项目，并尝试确定你的个人流程的地方。这些项目是困难的组合，但他们应该帮助你正式化你的流程，并找出适合你的东西。最重要的是，你应该着手分析你的工作方式和最适合你的东西。也许你没有执行我在本书建议的，关于个人发展的任何事情，但我希望你继续阅读这本书，并找到分析自己的方法。作为程序员，这样做会给你一种有效的方式，来成长和改进。

我们应该检查你至今为止所学到的知识，因为我会要求你尽可能多地应用它：

- 第二部分，你学会了如何 Hack，以及如何使你的起步尽可能顺利。
- 第三部分，你了解了数据结构和算法，但也学会了如何专注于质量和编写良好的测试。
- 第四部分，你将测试和质量技能应用于一些项目，专注于测试驱动开发和审计。
- 第五部分，你了解了解析，还有测量你的质量，在你工作和撰写有效测试的时候。
- 第六部分，你研究了 SQL 数据库，并学习了一个新的流程，用于分析数据并构建良好的数据。

第七部分，你会将一切东西应用于一系列的项目，确保关注三个方面的改进：

- 流程，通过尝试定义你的流程，并坚持它。
- 质量，通过专注于自动化测试，测试工具，并跟踪你的流程。
- 创造力，通过尝试解决没定义好的东西，并以一些松散、有趣的 Hack 开始。

你的流程是什么？

对于这本整本书，我已经告诉了你，我要让你使用什么流程工具。每个部分我都给你一个不同的挑战，专注于流程，质量或创造力，然后给你练习来做。你一直在跟踪你的质量，并从图表中查看什么适合你，什么不适合你。现在是时候开发自己的流程来完成一个项目，然后将其应用到本书这个部分的项目中。

花时间想出你的流程主题。它是否是 Hack 或者是 TDD 呢？是否始终是 TDD 并且有大量审计呢？它只是 Hack 和审计嘛？我的意思不是仅仅选择两件事，而应该考虑你的主题。把它当成你的个人风格来选择。我碰巧喜欢帽子和红衬衫。不要问我为什么，我只是喜欢他们。这

就是对你而言的流程描述。这是你夏日里的圆点连衣裙和黄色的鞋子。在编程中，我通常遵循“Hack，优化，测试，破坏”的主题。

一旦你有简单的主题声明，现在是时候为这个主题制定你的步骤了。将它们写在一张卡片上，以便你可以遵循他们，我会警告你，简单比复杂更好。复杂的流程很难处理。你的流程也应该命中创造力和质量。我的流程对于不同的项目是不同的，但是我在这本书中教导过你们。使用我至今为止教你的东西来想出自己的流程。

一旦你制定了你的流程，你可以回顾一下你的笔记，看看你能否找到指标，来证明你所选择的东西。也许你已经选择了 TDD，因为它让你觉得你写了更加稳固的代码，但是你在第五部分中的质量指标并不是非常好。对使用你喜欢的流程，我有一些要说的话，但是如果你喜欢的流程没有效果，那么现在就是把它扔进回收站的时候。

随着你的流程的出现，现在是时候来测试一些项目了。不要害怕犯错。有时，我们认为我们决定的东西是最好的，然后战火就像原子弹一样融化它。这是一个科学实验，所以如果一些事情是一场灾难，那么使用你的跟踪和指标，找出原因并简单地重新整备，再试一次。

练习 46： `blog`

> 原文：Exercise 46: blog
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

你应该按照本节开头的描述编写流程主题，你应该将流程列出并做好准备。为了起步，我们将为本节的其余部分提供一个名为 `blog` 的全新工具作为热身。

你应该慢慢参与这个项目，尽量不要赶时间。你的目标不是成为快速的程序员。通过系统地缓慢起步，你最好缓慢、流畅地建立速度，直到你的工作方式成为第二本能。如果你总是赶时间，那么你会粗心。

确保将你的笔记放在手边，并跟踪你的工作的实际情况和指标。你正在试图看看是否有一个适合你的流程，作为稍后的工作方式。并不是所有的方法一直有用，这就是为什么我试图教你各种工作策略，不同的程序员使用它们。如果你做这个项目，你发现你做的事情没有起作用，那么你的笔记将帮助你找出原因。在下一个项目中更改它，看看其它的东西是否效果更好。

挑战练习

我让你编写一个名为 `blog` 的简单的命令行博客工具。这是一个非常有创意的项目名称，用于一个有创意的项目。博客是一些早期程序员编写的第一个项目，但是你的项目将在本地生成博客，然后使用另一个名为 `rsync` 的工具将其发送到服务器。以下是此练习的要求：

- 如果你不知道什么是博客，那么你应该去创建一个，然后尝试一下。有很多平台，但你可能会喜欢 Wordpress 或 Tumblr。只需使用它一会儿，并记下你可能想要复制的功能。不要太疯狂了。
- 你将要学习如何使用模板系统来设计 HTML 页面。我建议你使用 mako 或 jinja 模板系统。这些系统允许你制作模板 HTML 文件，然后你可以根据用户放置在目录中的文本文件，将真实内容放入。
- 你将要使用 Markdown 作为你的博客格式，因此请为你的项目安装 markdown 库。
- 你的博客将是一个静态文件博客，因此你将需要使用 `python -m SimpleHTTPServer 8000`，就像 `SimpleHTTPServer` 指南中演示的那样。这会把转储目录中的文件提供给浏览器。
- 你需要一个名为 `blog` 的命令行工具，来处理他人的博客。
- 在你起步之前，请考虑你的博客工具所做的所有事情，然后设计所需的所有命令及其参数。然后查看 `docopt` 项目，来实现这些命令。

- 你应该使用 mock 来模拟你需要测试的东西，特别是错误情况。参考我在第三部分和第五部分中，我如何使用 `mock` 。
- 除此之外，你可以按照自己的意愿，随意开发这个 `blog` 工具。变得有创造力。所有要做的事情是，博客是以某种方式创建的，然后我可以放在要查看的服务器上。

最后，我会使用 `rsync` 将这样的博客放到网上，使用下列命令：

```
rsync -azv dist/* myserver.com:/var/www/myblog/
```

这可能更高级，但这可能是一个好时机，来学习如何部署静态文件。这里有一个研究性学习，也谈到如何使用亚马逊 S3 来实现它。

研究性学习

- 将静态文件部署到你自己的服务器是所有的事情，也很有趣，但如果 `blog` 工具适用于 Amazon S3 不是更好吗？有一个名为 `boto3` 的项目，将为你提供你所需的一切，使 `blog` 实现它。
- 编写一个 `blog serve` 命令，它使用 `SimpleHTTPServer` 类来简单部署博客，而不是单独生成博客。

练习 47：`bc`

> 原文：[Exercise 47: bc](#)
>
> 译者：[飞龙](#)
>
> 协议：[CC BY-NC-SA 4.0](#)
>
> 自豪地采用[谷歌翻译](#)

你应该热身并准备处理这个新项目。我通常假设，你将在一两天内的 2~3 小时的会话中完成这些项目，但你通常可以尽可能多地实现这些项目。

这个项目是，使用第五部分学到的内容，来为`bc`程序创建语言。我们已经在练习 36 中，为`bc`实现了简单的数学运算，但现在你将尽可能多地实现`bc`语言。`bc`大量运算符、函数和控制结构。你的目标是使用你对递归下降解析器的了解，来逐步实现它。

我将重点关注你的扫描器的构建，从扫描、解析、分析开始，并使用`bc`的样例代码进行测试。这个项目可能是巨大的，因为你手动实现语言，但尽可能多地完成语法。

挑战练习

`bc`语言不仅仅拥有处理数学运算的能力。我从不仅仅使用基础数学运算，完整的语言相当强大。你有能力定义函数，使用`if`语句，并实现许多其他常见的编程结构。在实现中，你无法实现整个`bc`语言，因为它太大了。相反，你应该实现这些东西：

- 所有的数学运算符
- 变量
- 函数
- `if`语句

这实际上是你应该实现该语言的顺序。首先，让运算符工作和解析良好。随意借鉴你在练习 35 中创建的简约实现，来起步。一旦实现了它，实现变量，这将需要使分析器正确处理变量的存储和检索。最后，你可以实现函数，然后是`if`语句。

你需要钻研 GNU 版本的`bc`的任何文档，因为它拥有语言的相当不错的完整描述，以便你可以实现它。它没有什么神奇的，因为他们大多从 C 复制一切，许多其他语言都类似于它。

当你处理这个挑战时，你需要花时间和步骤。实现语言的美妙之处是，你实际上可以以逻辑上清晰的顺序进行，从扫描到解析到分析，而不会在三个阶段之间有大的反弹。

最后，请记住，你正在实现一个递归下降解析器，实际上只是计算机科学解析中的低级版本。如果你正在做正经的解析工作，那么请使用一个解析器生成器，而不是用手写。用手编写它们只是一个有趣的挑战，并且是一种方法，来学习如何在逻辑上构建文本处理。

研究性学习

为了研究`bc`语言，你应该从 gnu.org 抓取源代码，并查找文件`bc.y`，`sbc.y`和`scan.l`。这可能令人困惑，所以去研究一个名为`lex`的工具，和一个名为`yacc`的工具。

练习 48： `ed`

> 原文：Exercise 48: ed
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

如果你的流程生效了，那么你应该能够专注于较长的项目，一次几个星期。在这个项目中，你的目标是创建尽可能精确的 `ed` 命令副本。这个练习的目标是不要有创意，而是要系统地实现另一个软件的精确副本。将其看作一个伪造的练习。你会做出一些非常好的东西，你可以用它代替原来的 `ed` ，没有人会知道。

这项工作是创建 `ed` 命令的“代码大师副本”，尽可能准确，也就是说，你的测试套件应该对真正的 `ed` 和你的版本运行同一脚本，来比较输出。这就像你在学习算法时所做的“代码大师副本”练习，除了你复制现有软件的行为，而不是试图记住它。这个流程是相似的，但你可以使用测试套件帮助它更快地进行。

挑战练习

`ed` 工具，是现存的最早的 Unix 文本编辑器之一，坦率地说，它非常糟糕。实际上我无法想像有人使用 `ed` 编辑文本，因为它是目前最受用户讨厌的软件之一。如果你无法想象，在 Unix 的黑暗时期，有多少计算机出了故障，制作一个 `ed` 的副本会让你见识一下。

`ed` 的一些概念是，虽然它支持脚本，但它最初是以交互方式使用的。就像用于文本文件的 MUD。你首先运行 `ed` ，它以命令模式启动，让你可以使用命令修改文本。当你执行需要输入的命令时，它将进入输入模式，直到该命令结束。你也必须知道行的地址来编辑它。这似乎是一种痛苦，但与其他文字编辑器相比，这是独角兽的魔法。

为了完成你的 `ed` 副本，你需要严重依赖于 Python 的 `re` 库，来使用正则表达式。我们在练习 31 中使用了这个库，所以通常你应该熟悉它和 RegEx。

我还建议你，在 45 分钟的会话内，尝试使用 `ed` ，来你的 `ed` 项目编写一些代码。这样做的痛苦将会教你如何复制它。

除此之外，你将需要阅读 `man ed` 页面来获取命令的基础知识，并可能会观看它的使用教程。一个很好的起步就是，在线寻找不同的示例脚本，并尝试将它们作为你的第一个测试用例。

> 注
>
> 我会给你一个线索，你需要使用 FSM 来处理 `ed` 命令的模态本质。

研究性学习

- 查找 GNU `ed` 的源代码，即使你不懂得 C 语言，也看一看它。
- 将你的 `ed` 实现变成一个模块，然后可以在其他项目中使用。你需要将其用于以后的练习。
- 永远不要再做这样的软件，除非你就是无聊了。

练习 49： `sed`

> 原文：Exercise 49: sed
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

当你学习如何制作快速而简陋的黑魔法时，你在练习 9 中实现了一个“低配版”的 `sed` 。在本练习中，您将尝试实现另一个精确的命令副本。在练习 48 的研究性学习中，您的任务是从您的 `ed` 实现中创建一个模块。如果你没有这样做，那么你需要为 `sed` 命令实现它，并且让 `sed` 使用它。

你的流程怎么样？你是否发现它在这些更长的项目上给了你帮助？有没有你认为需要改变的东西？你是否收集了指标，还是觉得你现在已经渡过了那个阶段？花点时间开始这个练习，看看你的日志，看看自从你开始读这本书以来，你已经改进了多少。

这个练习的挑战是，从练习 48 中的 `ed` 项目中获取代码，并在该项目中复用它。“可复用性”的概念对于软件的工作方式至关重要，但是很多时候，在项目中复用的计划会导致灾难。人们经常设计软件，以便每个组件可以在其他软件中使用，但是在这样做时，它们只是使设计过于复杂，并没有真正计划在任何其他项目中使用任何东西。最好使软件是离散的、独立的，然后取出可以用的一部分，在启动另一个项目时使用它。

我通常编写我的软件，不会担心可复用性。我不在乎项目的部分是否将用于其他项目。我只关心这一部分软件工作良好，并且质量高。当我启动一个新的项目，我会去看看我写的其他东西，看看有什么我可以再次使用的东西。如果我花时间在较旧的项目中拉出我可以使用的模块。这使我的复用流程看起来像这样：

- 通过自动化测试，实现功能完整的高质量软件。不要在乎用于其他软件的任何其它部分。
- 启动一个新项目，它可能使用另一个项目的代码。
- 回到第一个项目，将代码放入一个单独的模块，使这个第一个项目使用它，并且绝对不更改任何其它东西。
- 在适当的地方使用这个模块，一旦在原始的项目中，我的原始自动化测试运行良好，我就在新项目中使用该模块。
- 最后，尝试在新项目中使用新模块，会发现我需要在模块中进行的更改。我将进行更改，并确保它们与原始软件配合良好。

没有自动化测试的情况下，你不能做到它，所以如果你的 `ed` 项目没有测试，我想知道是否你读了这本书。回去，并确保你的测试完全覆盖了 `ed` 项目。

挑战练习

首先你要拉取`ed`项目的一部分，它处理命令，并把它放入`ed`所使用的模块中，而不会破坏测试。实话说，这将成为这个项目最难的部分之一，因为`sed`大多只是使用它们，而没有`ed`的交互界面的模态本质。

接下来，您将要从练习 9 中获取旧的代码，并重新启用它，或者在这个新项目上从零开始。一旦你确定了，你将开始使用ed模块实现尽可能多的sed。这项工作的创造力在于，确定两个项目需要使用什么，然后将其放在模块中。

这个实现中，你的目标是，制作一个`sed`命令的精确副本。这部分练习中没有创造力。只要试着尽可能小心，并使用自动化测试来确认，您的命令和原始的`sed`的工作方式是一样的。

最后，当您处理`sed`时，您将在模块中找到所需的东西。您需要对模块进行更改，使其在`sed`中工作，然后返回到`ed`并使其在那里工作。这三个项目之间的弹跳流程将是一个挑战，所以我建议你保持你的 45 分钟的时间块，所以你不会被上下文切换搞晕。

研究性学习

当你处理模块时，你是否发现了任何编程习惯，使拉取代码变得困难？它们是什么？

练习 50： `vi`

> 原文：Exercise 50: vi
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

你有一个模块，实现了 `ed` 和 `sed` 中使用的功能。显然，下一步是实现世界历史上最讨厌和实用的文本编辑器： `vi` 。如果你知道 Lisp 可以实现 Emacs，但没有人有时间创建一个伪装成文本编辑器的全新操作系统。人生苦短，不能整天按住三个按键并敲击 TAB 键。

这个练习的目的不是做一个非常精确的 `vi` 的副本。这是一个非常大的项目，但如果你想尝试它，那么就去做吧。你这个项目中的目标是，再次复用你的 `ed` 模块，并玩转 Python 的 `curse` 模块。 `curse` 模块让你处理旧式的文本终端窗口和图形操作。实际上“图形”应该加引号，因为 `curse` 中的实际图形很少。

你将使用 `curse` 来创建一个低级的 `vi` 实现，它可以打开文件，使用模块运行 `ed` 和 `sed` 命令，并使用 `curse` 将其显示到终端屏幕。你也会发现，尝试自动化测试它是非常困难的。如果你能弄清楚如何做一个假的 `curse` 测试框架，你会得到附加分，但是这将需要 Unix pty 系统的一些魔术技巧（我认为）。

使其可测试的更好方法是，将尽可能多的 `vi` 放入 Python 模块，以便你可以测试代码，而无需运行 `curses` 屏幕系统。当我说“模块”时，我并不意味着一个完整的 Python 模块，像使用 `ed` 模块一样，你可以使用 `pip` 安装它。我的意思是 `vi` 的代码中的模块，然后导入到你的项目中。

思考这个项目的方法是，将控制视图（ `curses` ）的代码与其余代码分开，以便你可以插入自己的假视图来测试。之后还剩下少量的功能，你可以通过实际运行你的 `vi` 来手动测试。

挑战练习

我们不会实现整个 `vi` 。我需要把它说清楚，因为实际的 `vi` 是旧的，非常复杂，所以实现一个完整的“代码大师副本”将需要很长时间。你实际上只做以下事情：

- 获取你的 `ed` 模块。
- 为其创建一个 `curses` UI。
- 使其在多个文件上工作。

这或多或少是你正在做的事情，所以你应该关注的第一件事是， `curses` 如何工作。阅读 `curses` 的文档，来了解它的工作原理，并根据需要编写尽可能多的测试来了解它。

一旦你掌握了`curses`，你就需要学习如何使用`vi`。我为这个练习包括一个`vi`速成课，你可以观看它，还有几个可以在线参考的`vi`速查表。我建议你看我的`vi`教程，并且在这个会话期间，尝试使用真正的`vi`来编辑代码。实际上从你的`ed`和`sed`的实现中，你可以了解`vi`如何工作。理论上，`vi`仅仅是图形化的`ed`，所以你几乎只是向`ed`提供一个更好的 UI。

研究性学习

- 你的`ed`实现中的有限状态机，如何与这个`vi`实现中的东西相匹配？假设你使用了这个设计。
- 实现 GUI 版本而不是`curses`，有多难？我不建议你这样做，但研究它，看看它需要什么。

练习 51：`lessweb`

> 原文：Exercise 51: lessweb
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

我们很接近这本书的末尾了，所以在最后两个练习中我将给你一个项目。你将要创建一个 Web 服务器。在本练习中，你只需了解 Python `http.server` 模块以及如何创建简单 Web 服务器来使用它。我将给你指示，然后让你阅读文档来了解如何实现它。这里没有太多的指导，因为现在你应该可以自己做大部分的事情。

创建 Web 服务器后，你将会编写一组测试，来尝试破坏你的 Web 服务器。我将在“破坏它”部分中为你提供一些指导，但现在你应该非常乐意在你编写的代码中找到缺陷。

挑战练习

你需要阅读 Python 3 的 `http.server` 文档来起步。你还应阅读 Python 3 的 `http.client` 文档以及 `requests` 的文档。你将使用 `requests` 或 `http.client` 为你创建的 `http.server` 编写测试。

接下来，你的工作是使用 `http.server` 创建一的 Web 服务器，可以执行以下操作：

- 从配置文件中进行配置
- 一直运行并处理收到的请求
- 提供配置目录中的文件
- 响应网站的请求并提供正确的内容
- 记录所有进入文件的请求，以便之后读取

如果你阅读文档中的示例，你大概可以以基本的方式，让大部分东西都工作。这个练习的一部分是，如何 Hack 一个朴素的 Web 服务器，所以你应该只是让它能够工作，然后我们将转到下一部分。

破坏它

你在本节中的工作是，以任何方式攻击你的 Web 服务器。你可以从 OWASP 十大漏洞列表开始，然后继续进行其他常见攻击。你还将阅读 Python 3 `os` 模块文档来实现一些修复。这是一个额外的错误列表，我敢肯定你会犯这些错误：

- 非预期的目录遍历。你可能从URL（`/some/file/index.html`）中获取基本路径，仅仅按照请求打开它。也许你在操作系统上添加了文件的完整路径（`/Users/zed/web/some/file/index.html`），并认为你做得很好。尝试使用 `..` 路径说明符来访问此目录外的文件。如果你可以请求 `/../../../../../../../../../etc/passwd`，那么你赢了。尝试解释为什么会发生这种情况，以及你可以如何解决这个问题。
- 没有处理非预期的请求。你最有可能寻找 `GET` 和 `POST`，但如果有人执行 `HEAD` 或 `OPTIONS`，会发生什么？
- 发送一个巨大的 HTTP 协议头。看看你是否可以通过发送一个非常大的 HTTP 请求头，使 Python `http.server` 崩溃或减慢速度。
- 请求未知域时不会出现错误。有些人认为它是一个功能（咳咳，Nginx），当服务器无法识别域时，会提供“随机”网站。你的服务器应该只是白名单，如果它不识别该域，它应该给出 404 错误。

这些只是人们所犯的一些小错误。研究尽可能多的其他人，然后为你的服务器编写自动化测试，以便在你解决问题之前展示它们。如果你的服务器中找不到任何这些错误，那么故意创建它们。了解如何犯下这些错误也是有益的。

研究性学习

阅读 Python 3 `os` 文档中的 `os.chroot` 函数。 研究如何使用这个函数和其他 `os` 模块的函数来创建“根目录限制”。 使用 `os` 中的许多函数以及你可以找到的任何模块，重写你的服务器，来正确地实现“根目录限制”，并丢弃权限变成安全用户（而不是 root）。在 Windows 上，这可能非常困难，所以要么在 Linux 计算机上尝试，要么完全跳过它。

练习 52： `moreweb`

> 原文：Exercise 52: moreweb
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

现在，你已经使用 Python `http.server` 库创建了一个 Web 服务器。你已经进行到最后一个项目了。你将使用你至今为止所学到的所有东西，从无到有创建你自己的 Web 服务器。在练习 51 中，你创建了大部分操作，它在 `http.server` 模块“上面”。你没有进行任何网络连接处理或 HTTP 协议解析。在最后的练习中，你将为你的 `lessweb` 服务器复制 `http.server` （所做的一切），并实现所有必要的零件。

挑战练习

为了完成此练习，你将需要阅读 Python 3 `asyncio` 模块的文档。这个库为你提供了工具，用于处理套接字请求，创建服务器，等待信号，以及大部分所需的其它东西。如果你想要一个额外的挑战，那么你可以使用 Python 3 `select` 模块，它提供了更低的级别的 API 来处理套接字。你应该使用此文档，来创建一系列小型套接字服务器和客户端。

一旦你了解如何创建通过 TCP/IP 套接字通话的服务器和客户端，则需要转而处理 HTTP 请求。该项目的这一部分将十分艰巨，因为 HTTP 标准丧心病狂，并且比其需要更复杂。我将从你可以设计的，最简单的 HTTP 解析库开始，然后用越来越多的样本进行扩展。第一个起始位置是 RFC 7230，但准备好体验一些人类搞出来的，最糟糕的写作。

研究 RFC 7230 的最佳方式是，首先提取“ABNF 汇总”附录中列出的所有语法。一眼看去，这似乎是疯狂的，因为这只是一个巨大的语法规范。你实际上在这本书的第五部分中，学到了如何阅读它，但是规模较小。你知道正则表达式，扫描器和解析器的工作原理，以及如何阅读这样的语法。所有你需要做的是研究这种语法，并一次实现一点。在实现它的时候，我将完全忽略任何“块”语法。

一旦你研究了这个语法，你应该开始为 HTTP 编写解析器，使用你已经创建的东西。使用你的数据结构，解析工具以及任何东西，来为 HTTP 的小型子集创建解析器。覆盖尽可能多的这种语法。为了帮助你，有一组测试文件，其中具有有效的 HTTP 请求，请访问 https://learncodethehardway.org/more-python-book/http_tests.zip。你可以下载这组测试用例，并通过你的解析器运行它们，来确保它有用。我从杰出的 And-HTTP 服务器中提取了许多这些测试用例，然后用更基本的例子来扩展它们。你的目标是使它们尽可能多地通过。

最后，一旦你有了一种方式，来编写一个良好的 `asyncio` 或者 `select` 套接字服务器，和一种解析 HTTP 的方式，你可以把它们放在一起，制作你的第一个带有功能的 Web 服务器。

破坏它

你一定要试图破坏这个 Web 服务器，但你也应该在这里尝试不同的东西。你已经编写了一个 HTTP 解析器，尝试使用 RDP 风格的解析器，以最合理的方式处理有效的 HTTP。你的解析器有很好的机会，来阻止许多不好的 HTTP 请求，所以找到一些以前的攻击，并在你的 Web 服务器上尝试它们。有几个网站上有自动化黑客工具，所以获取一个并将其对准你的服务器。但是要小心，并确保你只运行著名的测试工具，并且只在你自己的服务器上。

深入学习

如果你想完全了解 Web 服务器和技术，请使用你的 `moreweb` 服务器来创建 Web 框架。我建议先创建一个网站，然后从 Web 框架中提取出所需的模式。这种框架的目标是，封装你使用的模式，以便你可以简化后续的 Web 应用程序。与 `lessweb` 和 `moreweb` 的练习一样，你的目标也应该是研究，实现和利用 Web 框架的常见攻击。

如果你想深入 TCP/IP，我推荐 Jon C. Snader 的《Effective TCP/IP Programming》一书。这本书是用 C 语言写的，但它实际上是“笨办法学 TCP/IP》，涵盖 44 个主题，为你准备了简单的代码来了解基本的 TCP/IP 的工作原理。C 语言是 TCP/IP 的出生地，其他语言处理套接字连接的方式似乎很奇怪，直到你知道 C 语言是如何实现它的。通过研究它，你将会深入了解套接字服务器的工作原理。唯一的警告是这本书有点过时，所以代码应该工作，但它可能不是最新的代码。